Global guidance for local generalization in model checking

Citation for the original published paper (version of record):
Vediramana Krishnan, H., Chen, Y., Shoham, S. et al (2023). Global guidance for local generalization in model checking. Formal Methods in System Design, In Press.
http://dx.doi.org/10.1007/s10703-023-00412-3

Formal Methods in System Design
https://doi.org/10.1007/s10703-023-00412-3

Global guidance for local generalization in model checking

Hari Govind Vediramana Krishnan[1] · **YuTing Chen**[2] · **Sharon Shoham**[3] · **Arie Gurfinkel**[1]

Received: 5 January 2022 / Accepted: 25 January 2023

Abstract
SMT-based model checkers, especially IC3-style ones, are currently the most effective techniques for verification of infinite state systems. They infer *global* inductive invariants via *local* reasoning about a single step of the transition relation of a system, while employing SMT-based procedures, such as interpolation, to mitigate the limitations of local reasoning and allow for better generalization. Unfortunately, these mitigations intertwine model checking with heuristics of the underlying SMT-solver, negatively affecting stability of model checking. In this paper, we propose to tackle the limitations of locality in a systematic manner. We introduce explicit *global guidance* into the local reasoning performed by IC3-style algorithms. To this end, we extend the SMT-IC3 paradigm with three novel rules, designed to mitigate fundamental sources of failure that stem from locality. We instantiate these rules for Linear Integer Arithmetic and Linear Rational Aritmetic and implement them on top of Spacer solver in Z3. Our empirical results show that GSpacer, Spacer extended with global guidance, is significantly more effective than both Spacer and sole global reasoning, and, furthermore, is insensitive to interpolation.

Keywords Model checking · Constrained horn clauses · Interpolation · Automatic program verification

✉ Hari Govind Vediramana Krishnan
hgvk94@gmail.com

✉ YuTing Chen
yutingc@chalmers.se

✉ Sharon Shoham
sharon.shoham@gmail.com

✉ Arie Gurfinkel
arie.gurfinkel@uwaterloo.ca

[1] Department of Electrical and Computer Engineering, University of Waterloo, Waterloo, ON, Canada

[2] Chalmers University of Technology, Gothenburg, Sweden

[3] School of Computer Science, Tel Aviv University, Tel Aviv, Israel

Published online: 28 March 2023

1 Introduction

SMT-based Model Checking algorithms that combine SMT-based search for bounded counterexamples with interpolation-based search for inductive invariants are currently the most effective techniques for verification of infinite state systems. They are widely applicable, including for verification of synchronous systems, protocols, parameterized systems, and software.

The Achilles heel of these approaches is the mismatch between the *local* reasoning used to establish absence of bounded counterexamples and a *global* reason for absence of unbounded counterexamples (i.e., existence of an inductive invariant). This is particularly apparent in IC3-style algorithms [1], such as Spacer [2]. IC3-style algorithms establish bounded safety by repeatedly computing predecessors of error (or bad) states, blocking them by local reasoning about a single step of the transition relation of the system, and, later, using the resulting *lemmas* to construct a candidate inductive invariant for the global safety proof. The whole process is driven by the choice of local lemmas. Good lemmas lead to quick convergence, bad lemmas make even simple-looking problems difficult to solve.

The effect of local reasoning is somewhat mitigated by the use of interpolation in lemma construction. In addition to the usual inductive generalization by dropping literals from a blocked bad state, interpolation is used to further generalize the blocked state using theory-aware reasoning. For example, when blocking a bad state $x = 1 \wedge y = 1$, inductive generalization would infer a sub-clause of $x \neq 1 \vee y \neq 1$ as a lemma, while interpolation might infer $x \neq y$—a predicate that might be required for the inductive invariant. Spacer, that is based on this idea, is extremely effective, as demonstrated by its performance in CHC-COMP competitions [3]. The downside, however, is that the approach leads to a highly unstable procedure that is extremely sensitive to syntactic changes in the system description, changes in interpolation algorithms, and any algorithmic changes in the underlying SMT-solver.

An alternative approach, often called *invariant inference*, is to focus on the global safety proof, i.e., an inductive invariant. This has long been advocated by such approaches as Houdini [4], and, more recently, by a variety of machine-learning inspired techniques, e.g., FreqHorn [5], LinearArbitrary [6], and ICE-DT [7]. The key idea is to iteratively generate positive (i.e., reachable states) and negative (i.e., states that reach an error) examples and to compute a candidate invariant that separates these two sets. The reasoning is more focused towards the invariant, and, the search is restricted by either predicates, templates, grammars, or some combination. Invariant inference approaches are particularly good at finding simple inductive invariants. However, they do not generalize well to a wide variety of problems. In practice, they are often used to complement other SMT-based techniques.

In this paper, we present a novel approach that extends, what we call, *local reasoning* of IC3-style algorithms with *global guidance* inspired by the invariant inference algorithms described above. Our main insight is that the set of lemmas maintained by IC3-style algorithms hint towards a potential global proof. However, these hints are lost in existing approaches. We observe that letting the current set of lemmas, that represent candidate global invariants, guide local reasoning by introducing new lemmas and states to be blocked is often sufficient to direct IC3 towards a better global proof.

We present and implement our results in the context of Spacer–a solver for Constrained Horn Clauses (CHC)—implemented in the Z3 SMT-solver [8]. Spacer is used by multiple software model checking tools, performed remarkably well in CHC-COMP competitions

```
1  a, c := 0, 0;
2  // b, d := a, c;
3  b, d := 0, 0;
4  while(nd())
5  // inv: a - c = b - d;
6  {
7    if(nd()) { a++; b++; }
8    else { c++; d++; }
9  }
10 assert(a ≤ c ⇒ b ≤ d);
```

(a) myopic generalization

```
a, b := 0, 0;
while(nd())
// inv: a ≥ 0 ∧ b ≥ 0;
{
  a := a + b;
  b++;
}
assert(a ≥ 0);
```

(b) excessive generalization

```
a, b, c := 0, 0, 0;
while(nd())
// inv: b = c;
{
  a++; b++; c++;
}
assert(a ≥ 100 ⇒ b = c);
```

(c) stuck in a rut

Fig. 1 Verification tasks to illustrate sources of divergence for Spacer. The call *nd*() non-deterministically returns a Boolean value

[3], and is open-sourced. However, our results are fundamental and apply to any other IC3-style algorithm. While our implementation works with arbitrary CHC instances, we simplify the presentation by focusing on infinite state model checking of transition systems.

We illustrate the pitfalls of local reasoning using three examples shown in Fig. 1. All three examples are small, simple, and have simple inductive invariants. All three are challenging for Spacer. Where these examples are based on Spacer-specific design choices, each exhibits a fundamental deficiency that stems from local reasoning. We believe they can be adapted for any other IC3-style verification algorithm. The examples assume basic familiarity with the IC3 paradigm. Readers who are not familiar with it may find it useful to read the examples after reading Sect. 2.

Myopic generalization. Spacer diverges on the example in Fig. 1a by iteratively learning lemmas of the form $(a - c \leq k) \Rightarrow (b - d \leq k)$ for different values of k, where a, b, c, d are the program variables. These lemmas establish that there are no counterexamples of longer and longer lengths. However, the process never converges to the desired lemma $(a - c) \leq (b - d)$, which excludes counterexamples of any length. The lemmas are discovered using interpolation, based on proofs found by the SMT-solver. A close examination of the corresponding proofs shows that the relationship between $(a - c)$ and $(b - d)$ does not appear in the proofs, making it impossible to find the desired lemma by tweaking local interpolation reasoning. On the other hand, looking at the global proof (i.e., the set of lemmas discovered to refute a bounded counterexample), it is almost obvious that $(a - c) \leq (b - d)$ is an interesting generalization to try. Amusingly, a small, syntactic, but semantic preserving change of swapping Line 2 for Line 3 in Fig. 1a changes the SMT-solver proofs, affects local interpolation, and makes the instance trivial for Spacer.

Excessive (predecessor) generalization. Spacer diverges on the example in Fig. 1b by computing an infinite sequence of lemmas of the form $a + k_1 \times b \geq k_2$, where a and b are program variables, and k_1 and k_2 are integers. The root cause is excessive generalization in predecessor computation. The *Bad* states are $a < 0$, and their predecessors are states such as $(a = 1 \wedge b = -10)$, $(a = 2 \wedge b = -10)$, etc., or, more generally, regions $(a + b < 0)$, $(a + 2b < -1)$, etc. Spacer always attempts to compute the most general predecessor states. This is the best local strategy, but blocking these regions by learning their negation leads to the aforementioned lemmas. According to the global proof these lemmas do not converge to a linear invariant. An alternative strategy that under-approximates the problematic regions by (numerically) simpler regions and, as a result, learns simpler lemmas is desired (and is effective on this example). For example, region $a + 3b \leq -4$ can be

under-approximated by $a \leq 32 \wedge b \leq -12$, eventually leading to a lemma $b \geq 0$, that is a part of the final invariant: $(a \geq 0 \wedge b \geq 0)$.

Stuck in a rut. Finally, SPACER converges on the example in Fig. 1b, but only after unrolling the system for 100 iterations. During the first 100 iterations, SPACER learns that program states with $(a \geq 100 \wedge b \neq c)$ are not reachable because a is bounded by 1 in the first iteration, by 2 in the second, and so on. In each iteration, the global proof is updated by replacing a lemma of the form $a < k$ by lemma of the form $a < (k + 1)$ for different values of k. Again, the strategy is good locally – total number of lemmas does not grow and the bounded proof is improved. Yet, globally, it is clear that no progress is made since the same set of bad states are blocked again and again in slightly different ways. An alternative strategy is to abstract the literal $a \geq 100$ from the formula that represents the bad states, and, instead, conjecture that no states in $b \neq c$ are reachable.

Our approach: global guidance. As shown in the examples above, in all the cases that SPACER diverges, the missteps are not obvious locally, but are clear when the overall proof is considered. We propose three new rules, `Subsume`, `Concretize`, and, `Conjecture`, that provide global guidance, by considering existing lemmas, to mitigate the problems illustrated above. `Subsume` introduces a lemma that generalizes existing ones, `Concretize` under-approximates partially-blocked predecessors to focus on repeatedly unblocked regions, and `Conjecture` over-approximates a predecessor by abstracting away regions that are repeatedly blocked. The rules are generic, and apply to arbitrary SMT theories. Furthermore, we propose an efficient instantiation of the rules for the theory Linear Integer Arithmetic and the theory of Linear Rational Aritmetic.

We have implemented the new strategy, called GSPACER, in SPACER and compared it to the original implementation of SPACER. We show that GSPACER outperforms SPACER in benchmarks from CHC-COMP 2018, 2019, 2020, and 2021. More significantly, we show that the performance is independent of interpolation. While SPACER is highly dependent on interpolation parameters, and performs poorly when interpolation is disabled, the results of GSPACER are virtually unaffected by interpolation. We also compare GSPACER to LinearArbitrary [6], a tool that *infers invariants* using global reasoning. GSPACER outperforms LinearArbitrary on the benchmarks from [6]. These results indicate that global guidance mitigates the shortcomings of local reasoning.

The rest of the paper is structured as follows. Section 2 presents the necessary background. Section 3 introduces our *global guidance* as a set of abstract inference rules. Section 4 describes an instantiation of the rules to Linear Integer Arithmetic (LIA) and Linear Rational Arithmetic (LRA). Section 5 presents our empirical evaluation. Finally, Sect. 7 describes related work and concludes the paper.

This paper is an extended version of [9]. In comparison to [9], we have added additional empirical results on the effectiveness of GSPACER on LRA benchmarks and the effectiveness of individual global guidance rules. We also evaluate GSPACER on a wider class of benchmark problems.

2 Background

Logic. We consider first order logic modulo theories, and adopt the standard notation and terminology. A first-order language modulo theory $\mathcal{T}$ is defined over a signature Σ that consists of constant, function and predicate symbols, some of which may be *interpreted* by $\mathcal{T}$. As always, *terms* are constant symbols, variables, or function symbols applied to

terms; *atoms* are predicate symbols applied to terms; *literals* are atoms or their negations; *cubes* are conjunctions of literals; and *clauses* are disjunctions of literals. Unless otherwise stated, we only consider *closed* formulas (i.e., formulas without any free variables). As usual, we use sets of formulas and their conjunctions interchangeably.

MBP. Given a set of constants $\vec{v}$, a formula φ and a model $M \models \varphi$, Model Based Projection (MBP) of φ over the constants $\vec{v}$, denoted $\text{MBP}(\vec{v}, \varphi, M)$, computes a model-preserving under-approximation of φ projected onto $\Sigma \setminus \vec{v}$. That is, $\text{MBP}(\vec{v}, \varphi, M)$ is a formula over $\Sigma \setminus \vec{v}$ such that $M \models \text{MBP}(\vec{v}, \varphi, M)$ and any model $M' \models \text{MBP}(\vec{v}, \varphi, M)$ can be extended to a model $M'' \models \varphi$ by providing an interpretation for $\vec{v}$. There are polynomial time algorithms for computing MBP in Linear Arithmetic [2, 10]. We illustrate MBP in the following example:

Example 1 Consider the formula φ: $a' = a + 1 \wedge b' = b + 1 \wedge a' \leq 5 \wedge b' > 5$, and its model $M_1 = [a \mapsto 4, b \mapsto 5, a' \mapsto 5, b' \mapsto 6]$. Possible MBPs of φ relative to M_1 include:

$$\psi_1 \equiv a \leq 4 \wedge b > 4 \qquad \psi_2 \equiv a < b \wedge b = 5$$

Note that ψ_1 is equivalent to, and ψ_2 is a proper under-approximation of $\exists a', b' \cdot \varphi$, respectively. The soundness of our algorithms do not depend on the specific choice of MBP.

Interpolation. Given an unsatisfiable formula $A \wedge B$, an interpolant, denoted $\text{ITP}(A, B)$, is a formula I over the shared signature of A and B such that $A \Rightarrow I$ and $I \Rightarrow \neg B$. An interpolant need not be unique and may even contain terms that are not present in either A or B.

Example 2 Consider the formula A: $a = 0 \wedge b = 0 \wedge a' = a + 1 \wedge b' = b + 1$ and B: $a' < b' \wedge b' = 5$. Clearly, $A \wedge B$ is unsatisfiable. Two possible interpolants for $A \wedge B$ are $b' < 2$ and $a' = b'$. Both interpolants contain terms not present in the original formula.

Safety problem. A *transition system* is a pair $\langle Init, Tr \rangle$, where *Init* is a formula over Σ and *Tr* is a formula over $\Sigma \cup \Sigma'$, where $\Sigma' = \{s' \mid s \in \Sigma\}$.[1] The states of the system correspond to structures over Σ, *Init* represents the initial states and *Tr* represents the transition relation, where Σ is used to represent the pre-state of a transition, and Σ' is used to represent the post-state. For a formula φ over Σ, we denote by φ' the formula obtained by substituting each $s \in \Sigma$ by $s' \in \Sigma'$. A *safety problem* is a triple $\langle Init, Tr, Bad \rangle$, where $\langle Init, Tr \rangle$ is a transition system and *Bad* is a formula over Σ representing a set of bad states.

The safety problem $\langle Init, Tr, Bad \rangle$ has a *counterexample of length k* if the following formula is satisfiable: $Init^0 \wedge \bigwedge_{i=0}^{k-1} Tr^i \wedge Bad^k$, where φ^i is defined over $\Sigma^i = \{s^i \mid s \in \Sigma\}$ (a copy of the signature used to represent the state of the system after the execution of i steps) and is obtained from φ by substituting each $s \in \Sigma$ by $s^i \in \Sigma^i$, and Tr^i is obtained from *Tr* by substituting $s \in \Sigma$ by $s^i \in \Sigma^i$ and $s' \in \Sigma'$ by $s^{i+1} \in \Sigma^{i+1}$. The transition system is *safe* if the safety problem has no counterexample, of any length.

Inductive invariants. An *inductive invariant* is a formula *Inv* over Σ such that (i) $Init \Rightarrow Inv$, (ii) $Inv \wedge Tr \Rightarrow Inv'$, and (iii) $Inv \Rightarrow \neg Bad$. If such an inductive invariant exists, then the transition system is safe.

[1] In fact, a primed copy is introduced in Σ' only for the uninterpreted symbols in Σ. Interpreted symbols remain the same in Σ'.

Algorithm 1: SPACER algorithm as a set of guarded commands. We use the shorthand $\mathcal{F}(\varphi) = \mathcal{U}' \vee (\varphi \wedge Tr)$.

function SPACER:
In: $\langle Init, Tr, Bad\rangle$
Out: $\langle \text{SAFE}, Inv\rangle$ or UNSAFE
$Q := \emptyset$ // pob queue
$N := 0$ // maximum safe level
$\mathcal{O}_0 := Init, \mathcal{O}_i := \top$ **for all** $i > 0$ // lemma trace
$\mathcal{U} := Init$ // reachable states
forever do
`Candidate` $[\![\ \text{isSAT}(\mathcal{O}_N \wedge Bad)\ ]\!]\ Q := Q \cup \langle Bad, N\rangle$
`Predecessor` $[\![\ \langle\varphi, i+1\rangle \in Q,\ M \models \mathcal{O}_i \wedge Tr \wedge \varphi'\ ]\!]$
$Q := Q \cup \langle \text{MBP}(\vec{x}',\ Tr \wedge \varphi', M), i\rangle$
`Successor` $[\![\ \langle\varphi, i+1\rangle \in Q,\ M \models \mathcal{F}(\mathcal{U}) \wedge \varphi'\ ]\!]$
$\mathcal{U} := \mathcal{U} \vee \text{MBP}(\vec{x}, \mathcal{F}(\mathcal{U}), M)[\vec{x}' \mapsto \vec{x}]$
`Conflict` $[\![\ \langle\varphi, i+1\rangle \in Q,\ \mathcal{F}(\mathcal{O}_i) \Rightarrow \neg\varphi'\ ]\!]$
$\mathcal{O}_j := (\mathcal{O}_j \wedge \text{ITP}(\mathcal{F}(\mathcal{O}_i), \varphi')[\vec{x}' \mapsto \vec{x}])$ **for all** $j \leq i+1$
`Induction` $[\![\ \ell \in \mathcal{O}_{i+1}, \ell = (\varphi \vee \psi), \mathcal{F}(\varphi \wedge \mathcal{O}_i) \Rightarrow \varphi'\ ]\!]$
$\mathcal{O}_j := \mathcal{O}_j \wedge \varphi$ **for all** $j \leq i+1$
`Propagate` $[\![\ \ell \in \mathcal{O}_i, \mathcal{O}_i \wedge Tr \Rightarrow \ell'\ ]\!]\ \mathcal{O}_{i+1} := (\mathcal{O}_{i+1} \wedge \ell)$
`Unfold` $[\![\ \mathcal{O}_N \Rightarrow \neg Bad\ ]\!]\ N := N+1$
`Safe` $[\![\ \mathcal{O}_{i+1} \Rightarrow \mathcal{O}_i$ **for some** $i < N\ ]\!]$ **return** $\langle \text{SAFE}, \mathcal{O}_i\rangle$
`Unsafe` $[\![\ \text{isSAT}(Bad \wedge \mathcal{U})\ ]\!]$ **return** UNSAFE

Spacer. The safety problem defined above is an instance of a more general problem, CHC-SAT, of satisfiability of Constrained Horn Clauses (CHC). SPACER is a semi-decision procedure for CHC-SAT. However, to simplify the presentation, we describe the algorithm only for the particular case of the safety problem. We stress that SPACER, as well as the developments of this paper, apply to the more general setting of CHCs (both linear and non-linear). We assume that the only uninterpreted symbols in Σ are constant symbols, which we denote $\vec{x}$. Typically, these represent program variables. Without loss of generality, we assume that *Bad* is a cube.

We present SPACER in two ways: as a set of guarded commands (or rules) (Algorithm 1) and as a concrete implementation (Algorithm 2). We stress that the distinction between Algorithms 1 and 2 is in presentation only. Throughout the paper, we use SPACER to refer to either algorithm. First, we explain SPACER as a set of rules (Algorithm 1). It maintains the following. Current unrolling depth N at which a counterexample is searched (there are no counterexamples with depth less than N). A *trace* $\mathcal{O} = (\mathcal{O}_0, \mathcal{O}_1, \ldots)$ of *frames*, such that each frame $\mathcal{O}_i$ is a set of *lemmas*, and each lemma $\ell \in \mathcal{O}_i$ is a clause. A queue of *proof obligations* Q, where each proof obligation (POB) in Q is a pair $\langle\varphi, i\rangle$ of a cube φ and a level number i, $0 \leq i \leq N$. An under-approximation $\mathcal{U}$ of reachable states. Intuitively, each frame $\mathcal{O}_i$ is a candidate inductive invariant s.t. $\mathcal{O}_i$ over-approximates states reachable up to i steps from *Init*. The latter is ensured since $\mathcal{O}_0 = Init$, the trace is monotone, i.e., $\mathcal{O}_{i+1} \subseteq \mathcal{O}_i$, and each frame is inductive *relative* to its previous one, i.e., $\mathcal{O}_i \wedge Tr \Rightarrow \mathcal{O}'_{i+1}$. Each POB $\langle\varphi, i\rangle$ in Q corresponds to a suffix of a potential counterexample that has to be blocked in $\mathcal{O}_i$, i.e., has to be proven unreachable in i steps.

The `Candidate` rule adds an initial POB $\langle Bad, N\rangle$ to the queue. If a POB $\langle\varphi, i\rangle$ cannot be blocked because φ is reachable from frame $(i-1)$, the `Predecessor` rule

generates a predecessor ψ of φ using MBP and adds $\langle \psi, i-1 \rangle$ to Q. The `Successor` rule updates the set of reachable states if the POB is reachable. If the POB is blocked, the `Conflict` rule strengthens the trace $\mathcal{O}$ by using interpolation to learn a new lemma ℓ that blocks the POB, i.e., ℓ implies $\neg\varphi$. The `Induction` rule strengthens a lemma by inductive generalization and the `Propagate` rule pushes a lemma to a higher frame. If the *Bad* state has been blocked at N, the `Unfold` rule increments the depth of unrolling N. In practice, the rules are scheduled to ensure progress towards finding a counterexample. An implementation of the rules is shown in Algorithm 2 SPACER (Algorithm 2) starts off by adding *Bad* to the POB queue. Then, in a infinite loop, SPACER attempts to *block* all POBs in the POB queue (Line 4). To *block* a POB, SPACER first checks the precondition for both the `Predecessor` and `Conflict` rule (Line 6). If the POB is reachable from the previous frame, SPACER adds a predecessor POB to the queue (Line 7). SPACER then applies the `Unsafe` rule to see whether the reachable states intersect with *Bad* (Line 8). On the other hand, if the check at Line 6 fails, the POB is blocked at the current frame. In this case, Line 6 applies the `Conflict` rule to learn a lemma that blocks the POB (Line 10).

Algorithm 2: The SPACER algorithm.

```
   function GSpacer:
   In: ⟨Init, Tr, Bad⟩
   Out: An Inductive invariant or UNSAFE
   /* Initialize state of the solver                                   */
1  Q := ∅; N := 0; U := Init;
2  O_0 := Init; O_i := ⊤, ∀i > 0
3  Enqueue(Q, ⟨Bad, 0⟩)
4  while ⊤ do
5    ⟨φ, i⟩ := Pop(Q)
6    if IsSat(F(O_{i−1}) ∧ φ′) then
       // The pob φ cannot be blocked at i
7      AddPredecessor(⟨φ, i⟩)
8      if IsSat(U ∧ Bad) then return UNSAFE                  // Unsafe
9    else
       // The pob φ can be blocked at i
10     Block(⟨φ, i⟩)
11     for 0 ≤ j ≤ N do
12       for ℓ ∈ O_j \ O_{j+1} do
13         if O_j ∧ Tr ⇒ ℓ′ then
14           O_{j+1} := O_{j+1} ∧ ℓ                          // Propagate
15     if ∃0 ≤ j < N · O_j ⇒ O_{j−1} then
16       return ⟨SAFE, O_j⟩                                  // Safe
17     if O_N ⇒ ¬Bad then
18       N := N + 1                                          // Unfold
19       Push(Q, ⟨Bad, N⟩)
   function AddPredecessor:
20   if IsSat(F(U) ∧ φ′) then
21     find M_1 s.t M_1 ⊨ F(U) ∧ φ′
22     s := (MBP(x⃗, F(U), M_1)[x⃗′ ↦ x⃗])
23     U := U ∨ s                                            // Successor
25     return
26   find M_2 s.t M_2 ⊨ Θ
27   p := MBP(x⃗′, Tr ∧ φ′, M_2)
28   Push(Q, ⟨p, i − 1⟩)                                    // Predecessor
30   Push((Q, ⟨φ, i⟩))
   function Block:
31   ℓ_1 := ITP(F(O_{i−1}), φ′)                             // Conflict
32   ℓ_2 := α s.t. ℓ_1 = α ∨ β and F(O_{i−1} ∧ α) ⇒ α′       // Induction
33   for 0 ≤ j ≤ i do O_j := O_j ∧ ℓ_2
```

Fig. 2 An example of a simple program

```
1  a, b := 0, 0;
2  while(nd())
3  // inv: a = b;
4  {
5    a++;
6    b++;
7  }
8  assert(a ≤ 5 ⇒ b ≤ 5);
```

Example 3 As an illustrative example, we will explain how SPACER (Algorithm 2) checks the safety of the program in Fig. 2. For the program in Fig. 2, *Init* is $a = 0 \wedge b = 0$, *Tr* is $a' = a + 1 \wedge b' = b + 1$, and *Bad* states are $(a \leq 5 \wedge b > 5)$. Blocking *Bad* at levels 0 and 1 is straightforward. To block *Bad* at level 2, SPACER computes a predecessor using MBP. Let us assume MBP produces the POB $(a < b \wedge b = 5)$ (see Example 1). SPACER then checks whether the newly created POB $(a < b \wedge b = 5)$ is reachable at level 1. That is, SPACER checks ISSAT$(\mathit{Init} \wedge \mathit{Tr} \wedge a' < b' \wedge b' = 5)$ (Line 6). Since this is unsatisfiable, SPACER employs interpolation to learn a lemma. As we saw in Example 2, there are at least 2 possibilities for the interpolant.

If SPACER learns the lemma $a = b$, SPACER not only blocks the POB $(a < b \wedge b = 5)$ at level 1, it also *progpagates* the lemma $a = b$ (Line 14) and detects that it is inductive. Since $a = b$ is also enough to block *Bad*, SPACER returns SAFE (Line 16).

On the other hand, if SPACER learns the lemma $b < 2$, SPACER blocks the POB $(a < b \wedge b = 5)$ at level 1 but not at any higher levels. The lemma is not strong enough to block *Bad* at level 2. Therefore, in the next iteration, SPACER computes more predecessors of *Bad* at level 1 and attempts to block them.

Clearly, the lemmas that SPACER learns determine how fast it can prove the safety of the program. However, it is interpolation that decides which lemma SPACER learns. And interpolation is guided by the (local) heuristics inside the SMT solver.

3 Global guidance of local proofs

Algorithm 3: Global guidance rules for SPACER.

Subsume $[\![\ \mathcal{L}\subseteq\mathcal{O}_i, k\geq i, \mathcal{F}(\mathcal{O}_k)\Rightarrow\psi', \forall\ell\in\mathcal{L}.\psi\Rightarrow\ell\ ]\!]$
$\mathcal{O}_j := (\mathcal{O}_j\wedge\psi)$ **for all** $j\leq k+1$

Concretize $[\![\ \mathcal{L}\subseteq\mathcal{O}_i, \langle\varphi,j\rangle\in Q, \forall\ell\in\mathcal{L}.\text{ISSAT}(\varphi\wedge\neg\ell), \text{ISSAT}(\varphi\wedge\bigwedge\mathcal{L}), \gamma\Rightarrow\varphi, \text{ISSAT}(\gamma\wedge\bigwedge\mathcal{L})\ ]\!]$
$Q := Q\cup\langle\gamma, k+1\rangle$ **where** $k = \max\{j\,|\,\mathcal{O}_j\Rightarrow\neg\gamma\}$

Conjecture $[\![\ \mathcal{L}\subseteq\mathcal{O}_i, \langle\varphi,j\rangle\in Q, \varphi\equiv\alpha\wedge\beta, \forall\ell\in\mathcal{L}.\ell\Rightarrow\neg\beta\wedge\text{ISSAT}(\ell\wedge\alpha), \mathcal{U}\Rightarrow\neg\alpha\ ]\!]$
$Q := Q\cup\langle\alpha, k+1\rangle$ **where** $k = \max\{j\,|\,\mathcal{O}_j\Rightarrow\neg\alpha\}$

As illustrated by the examples in Fig. 1, while SPACER is generally effective, its local reasoning is easily confused. The effectiveness is very dependent on the local computation of predecessors using model-based projection, and lemmas using interpolation. In this section, we extend SPACER with three additional *global* reasoning rules. The rules are inspired

by the deficiencies illustrated by the motivating examples in Fig. 1. In this section, we present the rules abstractly, independent of any underlying theory, focusing on pre- and post-conditions. In Sect. 4, we specialize the rules for Linear Integer Arithmetic, and show how they are scheduled with the other rules of Spacer in an efficient verification algorithm. The new global rules are summarized in Algorithm 3. We use the same guarded command notation as in description of Spacer in Algorithm 1. Note that the rules supplement, and not replace, the ones in Algorithm 1.

Subsume is the most natural rule to explain. It says that if there is a set of lemmas $\mathcal{L}$ at level i, and there exists a formula ψ such that (a) ψ is stronger than every lemma in $\mathcal{L}$, and (b) ψ over-approximates states reachable in at most k steps, where $k \geq i$, then ψ can be added to the trace to subsume $\mathcal{L}$. This rule reduces the size of the global proof—that is, the number of total not-subsumed lemmas. Note that the rule allows ψ to be at a level k that is higher than i. The choice of ψ is left open. The details are likely to be specific to the theory involved. For example, when instantiated for LIA, `Subsume` is sufficient to solve example in Fig. 1a. Interestingly, `Subsume` is not likely to be effective for propositional IC3. In that case, ψ is a clause and the only way for it to be stronger than $\mathcal{L}$ is for ψ to be a syntactic sub-sequence of every lemma in $\mathcal{L}$, but such ψ is already explored by local inductive generalization (rule `Induction` in Algorithm 1).

Concretize applies to a pob, unlike `Subsume`. It is motivated by example in Fig. 1b that highlights the problem of excessive local generalization. Spacer always computes as general predecessors as possible. This is necessary for refutational completeness since in an infinite state system there are infinitely many potential predecessors. Computing the most general predecessor ensures that Spacer finds a counterexample, if it exists. However, this also forces Spacer to discover more general, and sometimes more complex, lemmas than might be necessary for an inductive invariant. Without a global view of the overall proof, it is hard to determine when the algorithm generalizes too much. The intuition for `Concretize` is that generalization is excessive when there is a single pob $\langle \varphi, j \rangle$ that is not blocked, yet, there is a set of lemmas $\mathcal{L}$ such that every lemma $\ell \in \mathcal{L}$ partially blocks φ. That is, for any $\ell \in \mathcal{L}$, there is a sub-region φ_ℓ of pob φ that is blocked by ℓ (i.e., $\ell \Rightarrow \neg\varphi_\ell$), and there is at least one state $s \in \varphi$ that is not blocked by any existing lemma in $\mathcal{L}$ (i.e., $s \models \varphi \wedge \bigwedge \mathcal{L}$). In this case, `Concretize` computes an under-approximation γ of φ that includes some not-yet-blocked state s. The new pob is added to the lowest level at which γ is not yet blocked. `Concretize` is useful to solve the example in Fig. 1b.

Conjecture guides the algorithm away from being stuck in the same part of the search space. A single pob φ might be blocked by a different lemma at each level that φ appears in. This indicates that the lemmas are too strong, and cannot be propagated successfully to a higher level. The goal of the `Conjecture` rule is to identify such a case to guide the algorithm to explore alternative proofs with a better potential for generalization. This is done by abstracting away the part of the pob that has been blocked in the past. The pre-condition for `Conjecture` is the existence of a pob $\langle \varphi, j \rangle$ such that φ is split into two (not necessarily disjoint) sets of literals, α and β. Second, there must be a set of lemmas $\mathcal{L}$, at a (typically much lower) level $i < j$ such that every lemma $\ell \in \mathcal{L}$ blocks φ, and, moreover, blocks φ by blocking β. Intuitively, this implies that while there are many different lemmas (i.e., all lemmas in $\mathcal{L}$) that block φ at different levels, all of them correspond to a *local* generalization of $\neg\beta$ that could not be propagated to block φ at higher levels. In this case, `Conjecture` abstracts the pob φ into α, hoping to generate an alternative way to block φ. Of course, α is conjectured only if it is not already blocked and does not contain any known reachable states. `Conjecture` is necessary for a quick convergence on the example in Fig. reffig:examplec. In some respect, `Conjecture` is akin to widening in Abstract

Interpretation [11]—it abstracts a set of states by dropping constraints that appear to prevent further exploration. Of course, it is also quite different since it does not guarantee termination. While `Conjecture` is applicable to propositional IC3 as well, it is much more significant in SMT-based setting since in many FOL theories a single literal in a POB might result in infinitely many distinct lemmas.

Each of the rules can be applied by itself, but they are most effective in combination. For example, `Concretize` creates less general predecessors, that, in the worst case, lead to many simple lemmas. At the same time, `Subsume` combines lemmas together into more complex ones. The interaction of the two produces lemmas that neither one can produce in isolation. At the same time, `Conjecture` helps unstuck the algorithm from a single unproductive POB, allowing the other rules to take effect.

4 Global guidance for linear arithmetic

In this section, we present a specialization of our general rules, shown in Algorithm 3, to the theory of Linear Integer Arithmetic (LIA) and Liner Rational Aritmetic (LRA). This requires solving two problems: identifying subsets of lemmas for pre-conditions of the rules (clearly using all possible subsets is too expensive), and applying the rule once its pre-condition is met. For lemma selection, we introduce a notion of syntactic clustering based on anti-unification. For rule application, we exploit basic properties of linear arithmetic for an effective algorithm. Our presentation is focused on linear arithmetic exclusively. However, the rules extend to combinations of linear arithmetic with other theories, such as the combined theory of LIA and Arrays.

In our presentation, we focus on specializing the rules to LIA. Adapting them to LRA is straightforward. The rest of this section is structured as follows. We begin with a brief background on LIA in Sect. 4.1. We then present our lemma selection scheme, which is common to all the rules, in Sect. 4.2, followed by a description of how the rules `Subsume` (in Sect. 4.3), `Concretize` (in Sect. 4.4), and `Conjecture` (in Sect. 4.5) are instantiated for LIA. We then explain an algorithm that integrates all the rules together in Sect. 4.6. Finally, we explain how to extend all the rules to LRA in Sect. 4.7.

4.1 Linear integer arithmetic: background

In the theory of Linear Integer Arithmetic (LIA), formulas are defined over a signature that includes interpreted function symbols $+, -, \times$, interpreted predicate symbols $<, \leq, |$, interpreted constant symbols $0, 1, 2, \ldots$, and uninterpreted constant symbols $a, b, \ldots, x, y, \ldots$. We write $\mathbb{Z}$ for the set of interpreted constant symbols, and call them *integers*. We use *constants* to refer exclusively to the uninterpreted constants (these are often called *variables* in LIA literature). Terms (and accordingly formulas) in LIA are restricted to be *linear*, that is, multiplication is never applied to two constants.

We write $\text{LIA}^{-\text{div}}$ for the fragment of LIA that excludes divisiblity $(d \mid h)$ predicates. A literal in $\text{LIA}^{-\text{div}}$ is a linear inequality; a cube is a conjunction of such inequalities, that is, a polytope. We find it convenient to use matrix-based notation for representing cubes in $\text{LIA}^{-\text{div}}$. A ground cube $c \in \text{LIA}^{-\text{div}}$ with p inequalities (literals) over k (uninterpreted) constants is written as $A \cdot \vec{x} \leq \vec{n}$, where A is a $p \times k$ matrix of coefficients in $\mathbb{Z}^{p \times k}$, $\vec{x} = (x_1 \cdots x_k)^T$ is a column vector that consists of the (uninterpreted) constants, and $\vec{n} = (n_1 \cdots n_p)^T$ is a column vector in $\mathbb{Z}^p$. For example, the cube $x \geq 2 \wedge 2x + y \leq 3$ is

written as $\begin{bmatrix}-1 & 0\\ 2 & 1\end{bmatrix} \cdot \begin{bmatrix}x\\ y\end{bmatrix} \leq \begin{bmatrix}-2\\ 3\end{bmatrix}$. In the sequel, all vectors are column vectors, super-script T denotes transpose, dot is used for a dot product and $[\vec{n}_1;\vec{n}_2]$ stands for a matrix of column vectors $\vec{n}_1$ and $\vec{n}_2$.

4.2 Lemma selection

A common pre-condition for all of our global rules in Algorithm 3 is the existence of a subset of lemmas $\mathcal{L}$ of some frame $\mathcal{O}_i$. Attempting to apply the rules for every subset of $\mathcal{O}_i$ is infeasible. In practice, we use syntactic similarity between lemmas as a predictor that one of the global rules is applicable, and restrict $\mathcal{L}$ to subsets of syntactically similar lemmas. In the rest of this section, we formally define what we mean by *syntactic similarity*, and how syntactically similar subsets of lemmas, called *clusters*, are maintained efficiently throughout the algorithm.

Syntactic similarity. A formula π with free variables is called a *pattern*. Note that we do not require π to be in LIA. Let σ be a substitution, i.e., a mapping from variables to terms. We write $\pi\sigma$ for the result of replacing all occurrences of free variables in π with their mapping under σ. A substitution σ is called *numeric* if it maps every variable to an integer, i.e., the range of σ is $\mathbb{Z}$. We say that a formula φ *numerically matches* a pattern π iff there exists a numeric substitution σ such that $\varphi = \pi\sigma$. Note that, as usual, the equality is syntactic. For example, consider the pattern $\pi = v_0 a + v_1 b \leq 0$ with free variables v_0 and v_1 and uninterpreted constants a and b. The formula $\varphi_1 = 3a + 4b \leq 0$ matches π via a numeric substitution $\sigma_1 = \{v_0 \mapsto 3, v_1 \mapsto 4\}$. However, $\varphi_2 = 4b + 3a \leq 0$, while semantically equivalent to φ_1, does not match π. $\varphi_3 = a + b \leq 0$ also does not match π because a and b do not have coefficients in φ_3, whereas π only matches literals with coefficients for both a and b.

Matching is extended to patterns in the usual way by allowing a substitution σ to map variables to variables. We say that a pattern π_1 is more general than a pattern π_2 if π_2 matches π_1. A pattern π is a *numeric anti-unifier* for a pair of formulas φ_1 and φ_2 if both φ_1 and φ_2 match π numerically. We write $\mathit{anti}(\varphi_1, \varphi_2)$ for a most general numeric anti-unifier of φ_1 and φ_2. We say that two formulas φ_1 and φ_2 are *syntactically similar* if there exists a numeric anti-unifier between them (i.e., $\mathit{anti}(\varphi_1, \varphi_2)$ is defined). Anti-unification is extended to sets of formulas in the usual way.

Clusters. We use anti-unification to define *clusters* of syntactically similar formulas. Let Φ be a fixed set of formulas, and π a pattern. A *cluster*, $\mathcal{C}_\Phi(\pi)$, is a subset of Φ such that every formula $\varphi \in \mathcal{C}_\Phi(\pi)$ numerically matches π. That is, π is a numeric anti-unifier for $\mathcal{C}_\Phi(\pi)$. In the implementation, we restrict the pre-conditions of the global rules so that a subset of lemmas $\mathcal{L} \subseteq \mathcal{O}_i$ is a cluster for some pattern π, i.e., $\mathcal{L} = \mathcal{C}_{\mathcal{O}_i}(\pi)$.

Clustering lemmas. We use the following strategy to efficiently keep track of available clusters. Let ℓ_{new} be a new lemma to be added to $\mathcal{O}_i$. Assume there is at least one lemma $\ell \in \mathcal{O}_i$ that numerically anti-unifies with ℓ_{new} via some pattern π. If such an ℓ does not belong to any cluster, a new cluster $\mathcal{C}_{\mathcal{O}_i}(\pi) = \{\ell_{\text{new}}, \ell\}$ is formed, where $\pi = \mathit{anti}(\ell_{\text{new}}, \ell)$. Otherwise, for every lemma $\ell \in \mathcal{O}_i$ that numerically matches ℓ_{new} and every cluster $\mathcal{C}_{\mathcal{O}_i}(\hat{\pi})$ containing ℓ, ℓ_{new} is added to $\mathcal{C}_{\mathcal{O}_i}(\hat{\pi})$ if ℓ_{new} matches $\hat{\pi}$, or a new cluster is formed using ℓ, ℓ_{new}, and any other lemmas in $\mathcal{C}_{\mathcal{O}_i}(\hat{\pi})$ that anti-unify with them. Note that a new lemma ℓ_{new} might belong to multiple clusters.

For example, suppose $\ell_{\text{new}} = (a \leq 6 \vee b \leq 6)$, and there is already a cluster $\mathcal{C}_{\mathcal{O}_i}(a \leq v_0 \vee b \leq 5) = \{(a \leq 5 \vee b \leq 5), (a \leq 8 \vee b \leq 5)\}$. Since ℓ_{new} anti-unifies with each of the lemmas in the cluster, but does not match the pattern $a \leq v_0 \vee b \leq 5$, a new cluster that includes all of them is formed w.r.t. a more general pattern: $\mathcal{C}_{\mathcal{O}_i}(a \leq v_0 \vee b \leq v_1) = \{(a \leq 6 \vee b \leq 6), (a \leq 5 \vee b \leq 5), (a \leq 8 \vee b \leq 5)\}$.

In the presentation above, we assumed that anti-unification is completely syntactic. This is problematic in practice since it significantly limits the applicability of the global rules. Recall, for example, that $a + b \leq 0$ and $2a + 2b \leq 0$ do not anti-unify numerically according to our definitions, and, therefore, do not cluster together. In practice, we augment syntactic anti-unification with simple rewrite rules that are applied greedily. For example, we normalize all LIA terms, take care of implicit multiplication by 1, and of associativity and commutativity of addition. In the future, it is interesting to explore how advanced anti-unification algorithms, such as [12, 13], can be adapted for our purpose.

4.3 Subsume rule for LIA

Recall that the `Subsume` rule (Algorithm 3) takes a cluster of lemmas $\mathcal{L} = \mathcal{C}_{\mathcal{O}_i}(\pi)$ and computes a new lemma ψ that subsumes all the lemmas in $\mathcal{L}$, that is $\psi \Rightarrow \bigwedge \mathcal{L}$. We find it convenient to dualize the problem. Let $\mathcal{S} = \{\neg \ell \mid \ell \in \mathcal{L}\}$ be the dual of $\mathcal{L}$, clearly $\psi \Rightarrow \bigwedge \mathcal{L}$ iff $(\bigvee \mathcal{S}) \Rightarrow \neg\psi$. Note that $\mathcal{L}$ is a set of clauses, $\mathcal{S}$ is a set of cubes, ψ is a clause, and $\neg\psi$ is a cube. In the case of $\text{LIA}^{-\text{div}}$, this means that $\bigvee \mathcal{S}$ represents a union of convex sets, and $\neg\psi$ represents a convex set that the `Subsume` rule must find. The strongest such $\neg\psi$ in $\text{LIA}^{-\text{div}}$ exists, and is the convex closure of $\mathcal{S}$. Thus, applying `Subsume` in the context of $\text{LIA}^{-\text{div}}$ is reduced to computing a convex closure of a set of (negated) lemmas in a cluster. Full LIA extends $\text{LIA}^{-\text{div}}$ with divisibility constraints. Therefore, `Subsume` obtains a stronger $\neg\psi$ by adding such constraints.

Example 4 For example, consider the following cluster:

$$\mathcal{L} = \{(x > 2 \vee x < 2 \vee y > 3), (x > 4 \vee x < 4 \vee y > 5), (x > 8 \vee x < 8 \vee y > 9)\}$$
$$\mathcal{S} = \{(x \geq 2 \wedge x \leq 2 \wedge y \leq 3), (x \geq 4 \wedge x \leq 4 \wedge y \leq 5), (x \geq 8 \wedge x \leq 8 \wedge y \leq 9)\}$$

The convex closure of $\mathcal{S}$ in $\text{LIA}^{-\text{div}}$ is $2 \leq x \leq 8 \wedge y \leq x + 1$. However, a stronger over-approximation exists in LIA: $2 \leq x \leq 8 \wedge y \leq x + 1 \wedge (2 \mid x)$.

In the sequel, we describe subsumeCube (Algorithm 4) which computes a cube φ that over-approximates $(\bigvee \mathcal{S})$. `Subsume` is then implemented by removing from $\mathcal{L}$ lemmas that are already subsumed by existing lemmas in $\mathcal{L}$, dualizing the result into $\mathcal{S}$, invoking subsumeCube on $\mathcal{S}$ and returning $\neg\varphi$ as a lemma that subsumes $\mathcal{L}$.

Recall that `Subsume` is tried only in the case $\mathcal{L} = \mathcal{C}_{\mathcal{O}_i}(\pi)$. We further require that the negated pattern, $\neg\pi$, is of the form $A \cdot \vec{x} \leq \vec{v}$, where A is a coefficients matrix, $\vec{x}$ is a vector of constants and $\vec{v} = (v_1 \cdots v_p)^T$ is a vector of p free variables. Under this assumption, $\mathcal{S}$ (the dual of $\mathcal{L}$) is of the form $\{(A \cdot \vec{x} \leq \vec{n}_i) \mid 1 \leq i \leq q\}$, where $q = |\mathcal{S}|$, and for each $1 \leq i \leq q$, $\vec{n}_i$ is a numeric substitution to $\vec{v}$ from which one of the negated lemmas in $\mathcal{S}$ is obtained. That is, $|\vec{n}_i| = |\vec{v}|$. In Example 4, $\neg\pi = x \leq v_1 \wedge -x \leq v_2 \wedge y \leq v_3$ and

$$A = \begin{bmatrix} 1 & 0 \\ -1 & 0 \\ 0 & 1 \end{bmatrix} \quad \vec{x} = \begin{bmatrix} x \\ y \end{bmatrix} \vec{v} = \begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix} \vec{n}_1 = \begin{bmatrix} 2 \\ -2 \\ 3 \end{bmatrix} \vec{n}_2 = \begin{bmatrix} 4 \\ -4 \\ 5 \end{bmatrix} \vec{n}_3 = \begin{bmatrix} 8 \\ -8 \\ 9 \end{bmatrix}$$

Each cube $(A \cdot \vec{x} \leq \vec{n}_i) \in \mathcal{S}$ is equivalent to $\exists \vec{v}. A \cdot \vec{x} \leq \vec{v} \wedge (\vec{v} = \vec{n}_i)$. Finally, $(\bigvee \mathcal{S}) \equiv \exists \vec{v}. (A \cdot \vec{x} \leq \vec{v}) \wedge (\bigvee (\vec{v} = \vec{n}_i))$. Thus, computing the over-approximation of $\mathcal{S}$ is reduced to (a) computing the convex hull H of a set of points $\{\vec{n}_i \mid 1 \leq i \leq q\}$, (b) computing divisibility constraints D that are satisfied by all the points, (c) substituting $H \wedge D$ for the disjunction in the equation above, and (c) eliminating variables $\vec{v}$. Both the computation of $H \wedge D$ and the elimination of $\vec{v}$ may be prohibitively expensive. We, therefore, over-approximate them. Our approach for doing so is presented in Algorithm 4, and explained in detail below.

Computing the convex hull of $\{\vec{n}_i \mid 1 \leq i \leq q\}$. Lines 3 to 8 compute the convex hull of $\{\vec{n}_i \mid 1 \leq i \leq q\}$ as a formula over $\vec{v}$, where variable v_j, for $1 \leq j \leq p$, represents the j^{th} coordinates in the vectors (points) $\vec{n}_i$. Some of the coordinates, v_j, in these vectors may be linearly dependent upon others. To simplify the problem, we first identify such dependencies and compute a set of linear equalities that expresses them (L in line 4). To do so, we consider a matrix $N_{q \times p}$, where the ith row consists of $\vec{n}_i^T$. The j^{th} column in N, denoted N_{*j}, corresponds to the j^{th} coordinate, v_j. The rank of N is the number of linearly independent columns (and rows). The other columns (coordinates) can be expressed by linear combinations of the linearly independent ones. To compute these linear combinations we use the kernel of $[N;\vec{1}]$ (N appended with a column vector of 1's), which is the set of all vectors $\vec{y}$ such that $[N;\vec{1}] \cdot \vec{y} = \vec{0}$, where $\vec{0}$ is the zero vector. Let $B = \text{kernel}([N;\vec{1}])$ be a basis for the kernel of $[N;\vec{1}]$. Then $|B| = p - \text{rank}(N)$, and for each vector $\vec{y} \in B$, the linear equality $[v_1 \cdots v_p\ 1] \cdot \vec{y} = 0$ holds in all the rows of N (i.e., all the given vectors satisfy it). We accumulate these equalities, which capture the linear dependencies between the coordinates, in L. Further, the equalities are used to compute $\text{rank}(N)$ coordinates (columns in N) that are linearly independent and, modulo L, uniquely determine the remaining coordinates. We denote by $\vec{v}^{L\downarrow}$ the subset of $\vec{v}$ that consists of the linearly independent coordinates. We further denote by $\vec{n}_i^{L\downarrow}$ the projection of $\vec{n}_i$ to these coordinates and by $N^{L\downarrow}$ the projection of N to the corresponding columns. We have that $(\bigvee (\vec{v} = \vec{n}_i)) \equiv L \wedge (\bigvee (\vec{v}^{L\downarrow} = \vec{n}_i^{L\downarrow}))$. In Example 4, the numeral matrix is $N = \begin{bmatrix} 2 & -2 & 3 \\ 4 & -4 & 5 \\ 8 & -8 & 9 \end{bmatrix}$, for which $\text{kernel}([N;\vec{1}]) = \{1\ 1\ 0\ 0^T,\ 1\ 0\ -1\ 1^T\}$. Therefore, L is the conjunction of equalities $v_1 + v_2 = 0 \wedge v_1 - v_3 + 1 = 0$, or, equivalently $v_3 = v_1 + 1 \wedge v_2 = -v_1$, $\vec{v}^{L\downarrow} = \begin{pmatrix} v_1 \end{pmatrix}^T$, and

$$\vec{n}_1^{L\downarrow} = \begin{bmatrix} 2 \end{bmatrix} \qquad \vec{n}_2^{L\downarrow} = \begin{bmatrix} 4 \end{bmatrix} \qquad \vec{n}_3^{L\downarrow} = \begin{bmatrix} 8 \end{bmatrix} \qquad N^{L\downarrow} = \begin{bmatrix} 2 \\ 4 \\ 8 \end{bmatrix}$$

Next, we compute the convex closure of $\bigvee (\vec{v}^{L\downarrow} = \vec{n}_i^{L\downarrow})$, and conjoin it with L to obtain H, the convex closure of $(\bigvee (\vec{v} = \vec{n}_i))$.

If the dimension of $\vec{v}^{L\downarrow}$ is one, as is the case in the example above, convex closure, C, of $\bigvee (\vec{v}^{L\downarrow} = \vec{n}_i^{L\downarrow})$ is obtained by bounding the sole element of $\vec{v}^{L\downarrow}$ based on its values in $N^{L\downarrow}$ (line 6). In Example 4, we obtain $C = 2 \leq v_1 \leq 8$.

If the dimension of $\vec{v}^{L_\downarrow}$ is greater than one, just computing the bounds of one of the constants is not sufficient. Instead, we use the concept of syntactic convex closure from [14] to compute the convex closure of $\bigvee(\vec{v}^{L_\downarrow} = \vec{n}_i^{L_\downarrow})$ as $\exists\vec{\alpha}.C$ where $\vec{\alpha}$ is a vector that consists of q fresh *rational* variables and C is defined as follows (line 8): $C = \vec{\alpha} \geq 0 \wedge \Sigma\vec{\alpha} = 1 \wedge \vec{\alpha}^T \cdot N^{L_\downarrow} = (\vec{v}^{L_\downarrow})^T$. C states that $(\vec{v}^{L_\downarrow})^T$ is a convex combination of the rows of $N^{L_\downarrow}$, or, in other words, $\vec{v}^{L_\downarrow}$ is a convex combination of $\{\vec{n}_i^{L_\downarrow} \mid 1 \leq i \leq q\}$.

To illustrate the syntactic convex closure, consider a second example with a set of cubes: $\mathcal{S} = \{(x \leq 0 \wedge y \leq 6), (x \leq 6 \wedge y \leq 0), (x \leq 5 \wedge y \leq 5)\}$. The coefficient matrix A, and the numeral matrix N are then: $A = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$ and $N = \begin{bmatrix} 0 & 6 \\ 6 & 0 \\ 5 & 5 \end{bmatrix}$. Here, kernel $([N;\vec{1}])$ is empty – all the columns are linearly independent, hence, $L = \mathit{true}$ and $\vec{v}^{L_\downarrow} = \vec{v}$. Therefore, syntactic convex closure is applied to the full matrix N, resulting in

$$\begin{aligned} C =& (\alpha_1 \geq 0) \wedge (\alpha_2 \geq 0) \wedge (\alpha_3 \geq 0) \wedge (\alpha_1 + \alpha_2 + \alpha_3 = 1) \\ & \wedge (6\alpha_2 + 5\alpha_3 = v_1) \wedge (6\alpha_1 + 5\alpha_3 = v_2) \end{aligned}$$

The convex closure of $\bigvee(\vec{v} = \vec{n}_i)$ is then $L \wedge \exists\vec{\alpha}.C$, which is $\exists\vec{\alpha}.C$ here.

Divisibility constraints. Inductive invariants for verification problems often require divisibility constraints. We, therefore, use such constraints, denoted D, to obtain a stronger over-approximation of $\bigvee(\vec{v} = \vec{n}_i)$ than the convex closure. To add a divisibility constraint for $v_j \in \vec{v}^{L_\downarrow}$, we consider the column $N_{*j}^{L_\downarrow}$ that corresponds to v_j in $N^{L_\downarrow}$. We find the largest positive integer d such that each integer in $N_{*j}^{L_\downarrow}$ leaves the same remainder when divided by d; namely, there exists $0 \leq r < d$ such that $n \bmod d = r$ for every $n \in N_{*j}^{L_\downarrow}$. This means that $d \mid (v_j - r)$ is satisfied by all the points $\vec{n}_i$. Note that such r always exists for $d = 1$. To avoid this trivial case, we add the constraint $d \mid (v_j - r)$ only if $d \neq 1$ (Line 4). We repeat this process for each $v_j \in \vec{v}^{L_\downarrow}$.

In Example 4, all the elements in the (only) column of the matrix $N^{L_\downarrow}$, which corresponds to v_1, are divisible by 2, and no larger d has a corresponding r. Thus, Line 12 of Algorithm 4 adds the divisibility condition $(2 \mid v_1)$ to D.

Eliminating existentially quantified variables using MBP. By combining the linear equalities exhibited by N, the convex closure of $N^{L_\downarrow}$ and the divisibility constraints on $\vec{v}$, we obtain $\exists\vec{\alpha}.L \wedge C \wedge D$ as an over-approximation of $\bigvee(\vec{v} = \vec{n}_i)$. Accordingly, $\exists\vec{v}.\exists\vec{\alpha}.\psi$, where $\psi = (A \cdot \vec{x} \leq \vec{v}) \wedge L \wedge C \wedge D$, is an over-approximation of $(\bigvee \mathcal{S}) \equiv \exists\vec{v}.(A \cdot \vec{x} \leq \vec{v}) \wedge (\bigvee(\vec{v} = \vec{n}_i))$ (Line 13). In order to get a LIA cube that overapproximates $\bigvee \mathcal{S}$, it remains to eliminate the existential quantifiers. Since quantifier elimination is expensive, and does not necessarily generate convex formulas (cubes), we approximate it using MBP. Namely, we obtain a cube φ that under-approximates $\exists\vec{v}.\exists\vec{\alpha}.\psi$ by applying MBP on ψ and a model $M_0 \models \psi$. We then use an SMT solver to drop literals from φ until it over-approximates $\exists\vec{v}.\exists\vec{\alpha}.\psi$, and hence also $\bigvee \mathcal{S}$ (lines 16 to 19). The result is returned by `Subsume` as an over-approximation of $\bigvee \mathcal{S}$.

Models M_0 that satisfy ψ and do not satisfy any of the cubes in $\mathcal{S}$ are preferred when computing MBP (line 14) as they ensure that the result of MBP is not subsumed by any of the cubes in $\mathcal{S}$.

Note that our final step is to do an MBP over the formula $\exists\vec{v}.\vec{\alpha}.(A \cdot \vec{x} \leq \vec{v}) \wedge L \wedge D \wedge C$. This requires MBP to support a mixture of integer and rational variables. To achieve this, we first push the quantifier over the rational variables inside; the formula becomes $\exists\vec{v}.(A \cdot \vec{x} \leq \vec{v}) \wedge L \wedge D \wedge \exists\vec{\alpha}.C)$. Then, we relax all constants in the constraints C to be rational and do an MBP over LRA to eliminate $\vec{\alpha}$. We then adjust the resulting formula back to integer arithmetic by multiplying each atom by the least common multiple of the denominators of the coefficients in it. Finally, we apply MBP over the integers to eliminate $\vec{v}$.

Considering Example 4 again, we get that $\psi = (x \le v_1) \wedge (-x \le v_2) \wedge (y \le v_3) \wedge (v_3 = 1 + v_1) \wedge (v_2 = -v_1) \wedge (2 \le v_1 \le 8) \wedge (2 \mid v_1)$ (the first three conjuncts correspond to $(A \cdot (x\ y)^T \le (v_1\ v_2\ v_3)^T)$). Note that in this case we do not have rational variables $\vec{\alpha}$ since $|\vec{v}^{L_\downarrow}| = 1$. Depending on the model, the result of MBP can be one of

$$y \le x + 1 \wedge 2 \le x \le 8 \wedge (2 \mid y - 1) \wedge (2 \mid x) \qquad x \ge 2 \wedge x \le 2 \wedge y \le 3$$
$$y \le x + 1 \wedge 2 \le x \le 8 \wedge (2 \mid x) \qquad x \ge 8 \wedge x \le 8 \wedge y \le 9$$
$$y \ge x + 1 \wedge y \le x + 1 \wedge 3 \le y \le 9 \wedge (2 \mid y - 1)$$

However, we prefer a model that does not satisfy any cube in $\mathcal{S} = \{(x \ge 2 \wedge x \le 2 \wedge y \le 3), (x \le 4 \wedge x \ge 4 \wedge y \le 5), (x \le 8 \wedge x \ge 8 \wedge y \le 9)\}$, rules off the two possibilities on the right. None of these cubes cover ψ, hence generalization is used.

If the first cube is obtained by MBP, it is generalized into $y \le x + 1 \wedge x \ge 2 \wedge x \le 8 \wedge (2 \mid x)$; the second cube is already an over-approximation; the third cube is generalized into $y \le x + 1 \wedge y \le 9$. Indeed, each of these cubes over-approximates $\bigvee \mathcal{S}$.

Algorithm 4: An implementation of the Subsume rule for the dual of a cluster $\mathcal{S} = \{A \cdot \vec{x} \le \vec{n}_i \mid 1 \le i \le q\}$.

function SUBSUMECUBE:
In: $\mathcal{S} = \{(A \cdot \vec{x} \le \vec{n}_i) \mid 1 \le i \le q\}$,
Out: An over-approximation of $(\bigvee \mathcal{S})$.
/* $\vec{v}$ are integer variables such that: $(\bigvee \mathcal{S}) \iff \exists \vec{v}.\, (A \cdot \vec{x} \le \vec{v}) \wedge (\bigvee \vec{v} = \vec{n}_i)$ */
$N := [\vec{n}_1; \cdots; \vec{n}_q]^T$
/* Compute the set of linear dependencies implied by N */
$B := \text{kernel}([N;\ \vec{1}])$
$L := \bigwedge_{\vec{y} \in B} (\, v_1 \ \cdots \ v_p \ 1\,) \cdot \vec{y} = 0$
if $|\vec{v}^{L_\downarrow}| = 1$ **then**
// Convex closure over a single constant
$v_i \in \vec{v}^{L_\downarrow}$
$C := \min(N_{*i}) \le v_i \le \max(N_{*i})$
else
// Syntactic convex closure
$C := (\vec{\alpha}^T \cdot N^{L_\downarrow} = (\vec{v}^{L_\downarrow})^T) \wedge (\Sigma \vec{\alpha} = 1) \wedge (\vec{\alpha} \ge 0)$
/* Compute divisibility constraints */
$D := \top$
for $v_j \in \vec{v}^{L_\downarrow}$ **do**
if $\exists d, r.\, d \ne 1 \wedge (\forall n \in N^{L_\downarrow}_{*j}.\, (n \bmod d = r))$ **then**
$D := D \wedge d \mid (v_j - r)$
$\psi := (A \cdot \vec{x} \le \vec{v}) \wedge L \wedge C \wedge D$
/* Under-approximate quantifier elimination */
find M_0 s.t. $M_0 \models \psi$ and, if possible, $M_0 \not\models (\bigvee S)$
$\varphi := \text{MBP}((\vec{\alpha}\ \vec{v}), \psi, M_0)$
/* Over-approximate quantifier elimination */
while ISSAT$(\neg\varphi \wedge \psi)$ **do**
find M_1 s.t. $M_1 \models (\neg\varphi \wedge \psi)$
$\varphi := \bigwedge \{\ell \in \varphi \mid \neg(M_1 \models \neg\ell)\}$
return φ

Algorithm 5: An implementation of the Concretize rule in LIA.

function CONCRETIZE:
In: A POB $\langle \varphi, j \rangle$ in LIA$^{-\text{div}}$, a cluster of LIA$^{-\text{div}}$ lemmas $\mathcal{L} = \mathcal{C}_{\mathcal{O}_i}(\pi)$ s.t. π is non-linear, ISSAT$(\varphi \wedge \bigwedge \mathcal{L})$
Out: A cube γ such that $\gamma \Rightarrow \varphi$ and $\forall \ell \in \mathcal{L}.$ ISSAT$(\gamma \wedge \ell)$
$U := \{x \mid \text{COEFF}(x, \pi) \in \text{VARS}(\pi)\}$
find M s.t. $M \models \varphi \wedge \bigwedge \mathcal{L}$
$\gamma := \top$
foreach $lit \in \varphi$ **do**
if CONSTS$(lit) \cap U \ne \emptyset$ **then**
$\gamma := \gamma \wedge$ CONCRETIZE_LIT(lit, M, U)
else $\gamma := \gamma \wedge lit$
$\gamma :=$ RM_SUBSUME(γ)
return γ
function CONCRETIZE_LIT:
In: A literal $lit = \Sigma_i n_i x_i \le b_j$ in LIA$^{-\text{div}}$, model $M \models lit$, and a set of constants U
Out: A cube γ^{lit} that concretizes lit
/* Construct a single literal using all the constants in CONSTS$(lit) \setminus U$ */
$\gamma^{lit} := \emptyset$
$s := 0$
foreach $x_i \in$ CONSTS$(lit) \setminus U$ **do**
$s := s + n_i x_i$
$\gamma^{lit} := (s \le M[s])$
/* Generate one dimensional literals for each constant in U */
foreach $x_i \in$ CONSTS$(lit) \cap U$ **do**
$\gamma^{lit} := \gamma^{lit} \wedge (n_i x_i \le M[n_i x_i])$
return γ^{lit}

4.4 Concretize rule for LIA

The `Concretize` rule (Algorithm 3) takes a cluster of lemmas $\mathcal{L} = \mathcal{C}_{\mathcal{O}_i}(\pi)$ and a POB $\langle \varphi, j \rangle$ such that each lemma in $\mathcal{L}$ partially blocks φ, and creates a new POB γ that is still not blocked by $\mathcal{L}$, but γ is more concrete, i.e., $\gamma \Rightarrow \varphi$. In our implementation, this rule is applied when φ is in $\text{LIA}^{-\text{div}}$. We further require that the pattern, π, of $\mathcal{L}$ is non-linear, i.e., some of the constants appear in π with free variables as their coefficients. We denote these constants by U. An example is the pattern $\pi = v_0 x + v_1 y + z \leq 0$, where $U = \{x, y\}$. Having such a cluster is an indication that attempting to block φ in full with a single lemma may require to track non-linear correlations between the constants, which is impossible to do in LIA. In such cases, we identify the coupling of the constants in U in POBs (and hence in lemmas) as the potential source of non-linearity. Hence, we concretize (strengthen) φ into a POB γ where the constants in U are no longer coupled to any other constant.

Coupling. Formally, constants u and v are *coupled* in a cube c, denoted $u \bowtie_c v$, if there exists a literal *lit* in c such that both u and v appear in *lit* (i.e., their coefficients in *lit* are non-zero). For example, x and y are coupled in $x + y \leq 0 \wedge z \leq 0$ whereas neither of them are coupled with z. A constant u is said to be *isolated* in a cube c, denoted $\text{ISO}(u, c)$, if it appears in c but it is not coupled with any other constant in c. In the above cube, z is isolated.

Concretization by decoupling. Given a POB φ (a cube) and a cluster $\mathcal{L}$, Algorithm 5 presents our approach for concretizing φ by decoupling the constants in U — those that have variables as coefficients in the pattern of $\mathcal{L}$ (line 2). Concretization is guided by a model $M \models \varphi \wedge \bigwedge \mathcal{L}$, representing a part of φ that is not yet blocked by the lemmas in $\mathcal{L}$ (line 3). Given such M, we concretize φ into a *model-preserving* under-approximation that isolates all the constants in U and preserves all other couplings. That is, we find a cube γ, such that

$$\gamma \Rightarrow \varphi \quad M \models \gamma \quad \forall u \in U.\text{ISO}(u, \gamma) \quad \forall u, v \notin U.(u \bowtie_\varphi v) \Rightarrow (u \bowtie_\gamma v) \tag{1}$$

Note that γ is not blocked by $\mathcal{L}$ since M satisfies both $\bigwedge \mathcal{L}$ and γ. For example, if $\varphi = (x + y \leq 0) \wedge (x - y \leq 0) \wedge (x + z \geq 0)$ and $M = [x = 0, y = 0, z = 1]$, then $\gamma = 0 \leq y \leq 0 \wedge x \leq 0 \wedge x + z \geq 1$ is a model preserving under-approximation that isolates $U = \{y\}$.

Algorithm 5 computes such a cube γ by a point-wise concretization of the literals of φ followed by the removal of subsumed literals. Literals that do not contain constants from U remain unchanged. A literal of the form $lit = t \leq b$, where $t = \sum_i n_i x_i$ (recall that every literal in $\text{LIA}^{-\text{div}}$ can be normalized to this form), that includes constants from U is concretized into a *cube* by (1) isolating each of the summands $n_i x_i$ in t that include U from the rest, and (2) for each of the resulting sub-expressions creating a literal that uses its value in M as a bound. Formally, t is decomposed to $s + \sum_{x_i \in U} n_i x_i$, where $s = \sum_{x_i \notin U} n_i x_i$. The concretization of *lit* is the cube $\gamma^{lit} = s \leq M[s] \wedge \bigwedge_{x_i \in U} n_i x_i \leq M[n_i x_i]$, where $M[t']$ denotes the interpretation of t' in M. Note that $\gamma^{lit} \Rightarrow lit$ since the bounds are stronger than the original bound on t: $M[s] + \sum_{x_i \in U} M[n_i x_i] = M[t] \leq b$. This ensures that γ, obtained

by the conjunction of literal concretizations, implies φ. It trivially satisfies the other conditions of Eq. (1).

For example, the concretization of the literal $(x + y \leq 0)$ with respect to $U = \{y\}$ and $M = [x = 0, y = 0, z = 1]$ is the cube $x \leq 0 \wedge y \leq 0$. Applying concretization in a similar manner to all the literals of the cube $\varphi = (x + y \leq 0) \wedge (x - y \leq 0) \wedge (x + z \geq 0)$ from the previous example, we obtain the concretization $x \leq 0 \wedge 0 \leq y \leq 0 \wedge x + z \geq 0$. Note that the last literal is not concretized as it does not include y.

4.5 Conjecture rule for LIA

The `Conjecture` rule (see Algorithm 3) takes a set of lemmas $\mathcal{L}$ and a POB $\varphi \equiv \alpha \wedge \beta$ such that all lemmas in $\mathcal{L}$ block β, but none of them blocks α, where α does not include any known reachable states. It returns α as a new POB.

For LIA, `Conjecture` is applied when the following conditions are met:

(1) the POB φ is of the form $\varphi_1 \wedge \varphi_2 \wedge \varphi_3$, where $\varphi_3 = (\vec{n}^T \cdot \vec{x} \leq b)$, and φ_1 and φ_2 are any cubes. The sub-cube $\varphi_1 \wedge \varphi_2$ acts as α, while the sub-cube $\varphi_2 \wedge \varphi_3$ acts as β.
(2) The cluster $\mathcal{L}$ consists of $\{bg \vee (\vec{n}^T \cdot \vec{x} \geq b_i) \mid 1 \leq i \leq q\}$, where $b_i > b$ and $bg \Rightarrow \neg\varphi_2$. This means that each of the lemmas in $\mathcal{L}$ blocks $\beta = \varphi_2 \wedge \varphi_3$, and they may be ordered as a sequence of increasingly stronger lemmas, indicating that they were created by trying to block the POB at different levels, leading to too strong lemmas that failed to propagate to higher levels.
(3) The formula $(bg \vee (\vec{n}^T \cdot \vec{x} \geq b_i)) \wedge \varphi_1 \wedge \varphi_2$ is satisfiable, that is, none of the lemmas in $\mathcal{L}$ block $\alpha = \varphi_1 \wedge \varphi_2$, and
(4) $\mathcal{U} \Rightarrow \neg(\varphi_1 \wedge \varphi_2)$, that is, no state in $\varphi_1 \wedge \varphi_2$ is known to be reachable.

If all four conditions are met, we conjecture $\alpha = \varphi_1 \wedge \varphi_2$. This is implemented by CONJECTURE, that returns α (or $\perp$ when the pre-conditions are not met).

For example, consider the POB $\varphi = x \geq 10 \wedge (x + y \geq 10) \wedge y \leq 10$ and a cluster of lemmas $\mathcal{L} = \{(x + y \leq 0 \vee y \geq 101), (x + y \leq 0 \vee y \geq 102)\}$. In this case, $\varphi_1 = x \geq 10$, $\varphi_2 = (x + y \geq 10)$, $\varphi_3 = y \leq 10$, and $bg = x + y \leq 0$. Each of the lemmas in $\mathcal{L}$ block $\varphi_2 \wedge \varphi_3$ but none of them block $\varphi_1 \wedge \varphi_2$. Therefore, we conjecture $\varphi_1 \wedge \varphi_2$: $x \geq 10 \wedge (x + y \geq 10)$.

4.6 Putting it all together

Having explained the implementation of the new rules for LIA, we now put all the ingredients together into an algorithm, GSPACER. In particular, we present our choices as to when to apply the new rules, and on which clusters of lemmas and POBS. As can be seen in Sect. 5, this implementation works very well on a wide range of benchmarks.

Algorithm 6: GSPACER for LIA.

```
1  function GSpacer:
   In: ⟨Init, Tr, Bad⟩
   Out: An Inductive invariant or UNSAFE
   /* Initialize state of the solver */
2  Q := ∅; N := 0; U := Init;
3  O_0 := Init; O_i := ⊤, ∀i > 0
4  Enqueue(Q, ⟨Bad, 0⟩)
5  while ⊤ do
6   ⟨φ, i⟩ := Pop(Q)
7   if ConcretizePOB(⟨φ, i⟩) = ⊤ then
8    continue
9   if isSat(F(O_{i−1}) ∧ φ′) then
     // pob φ cannot be blocked at i
10   AddPredecessor(⟨φ, i⟩)
11   if isSat(U ∧ Bad) then
12    return UNSAFE                    // Unsafe
13  else
     // The pob φ can be blocked at i
14   Block(⟨φ, i⟩)
15   for 0 ≤ j ≤ N do
16    for ℓ ∈ O_j \ O_{j+1} do
17     if O_j ∧ Tr ⇒ ℓ′ then
18      O_{j+1} := O_{j+1} ∧ ℓ   // Propagate
19   if ∃0 ≤ j < N · O_j ⇒ O_{j−1} then
20    return ⟨SAFE, O_j⟩                // Safe
21   if O_N ⇒ ¬Bad then
22    N := N + 1                        // Unfold
23    push(Q, ⟨Bad, N⟩)

24 function ConcretizePOB:
25  ⟨π_1, L_1⟩ := C_pob(⟨φ, i⟩)
26  L_2 := {ℓ | ℓ ∈
     L_1 ∧ isSat(ℓ ∧ φ) ∧ isSat(¬ℓ ∧ φ)}
27  if (L_2 ≠
     ∅ ∧ nonLin(π_1) ∧ isSat(⋀ L_2 ∧ φ))
     then
28   γ := Concretize(φ, ⟨π_1, L_2⟩)
29   k := max{j | O_j ⇒ ¬γ}
30   push(Q, ⟨γ, k⟩)              // Concretize
31   push(Q, ⟨φ, i⟩)
32   return ⊤
33  else return ⊥
34 function AddPredecessor:
35  if isSat(F(U) ∧ φ′) then
36   find M_1 s.t M_1 ⊨ F(U) ∧ φ′
37   s := (MBP(x⃗, F(U), M_1)[x⃗′ ↦ x⃗])
38   U := U ∨ s                        // Successor
39   return
40  find M_2 s.t M_2 ⊨ Θ
41  p := MBP(x⃗′, Tr ∧ φ′, M_2)
42  push(Q, ⟨p, i − 1⟩)          // Predecessor
43  push(Q, ⟨φ, i⟩)
44 function Block:
45  ℓ := GEN(F(O_{i−1}), φ′)     // Conflict
46  for 0 ≤ j ≤ i do O_j := O_j ∧ ℓ
47  ⟨π_3, L_3⟩ = C_lemma(ℓ)
48  α := Conjecture(φ, L_3, U)
49  if α ≠ ⊥ then
50   k := max{j | O_j ⇒ ¬α}
51   push(Q, ⟨α, k⟩)              // Conjecture
52  if ¬π_3 = A · x⃗ ≤ v⃗ then
53   ψ := subsume(⟨π_3, L_3⟩)
54   k := max{j | F(O_j) ⇒ ψ′}
55   O_j := O_j ∧ ψ for all j ≤ k + 1  // Subsume
```

Algorithm 6 presents GSPACER. The comments to the right side of a line refer to the abstract rules in Algorithms 1 and 3. Just like SPACER, GSPACER iteratively computes predecessors (Line 10) and blocks them (Line 14) in an infinite loop. Whenever a POB is proven to be reachable, the reachable states are updated (line 38). If *Bad* intersects with a reachable state, GSPACER terminates and returns UNSAFE (line 12). If one of the frames is an inductive invariant, GSPACER terminates with SAFE (line 20).

When a POB $\langle \varphi, i \rangle$ is handled, we first apply the `Concretize` rule, if possible (Line 7). Recall that CONCRETIZE (Algorithm 5) takes as input a cluster that partially blocks φ and has a non-linear pattern. To obtain such a cluster, we first find, using $\mathcal{C}_{pob}(\langle \varphi, i \rangle)$, a cluster $\langle \pi_1, \mathcal{L}_1 \rangle = \mathcal{C}_{\mathcal{O}_k}(\pi_1)$, where $k \leq i$, that includes *some* lemma (from frame k) that blocks φ; if none exists, $\mathcal{L}_1 = \emptyset$. We then filter out from $\mathcal{L}_1$ lemmas that completely block φ as well as lemmas that are irrelevant to φ, i.e., we obtain $\mathcal{L}_2$ by keeping only lemmas that partially block φ. We apply CONCRETIZE on $\langle \pi_1, \mathcal{L}_2 \rangle$ to obtain a new POB that under-approximates φ

if (1) the remaining sub-cluster, $\mathcal{L}_2$, is non-empty, (2) the pattern, π_1, is non-linear, and (3) $\bigwedge \mathcal{L}_2 \wedge \varphi$ is satisfiable, i.e., a part of φ is not blocked by any lemma in $\mathcal{L}_2$.

Once a POB is blocked, and a new lemma that blocks it, ℓ, is added to the frames, an attempt is made to apply the `Subsume` and `Conjecture` rules on a cluster that includes ℓ. To that end, the function $\mathcal{C}_{lemma}(\ell)$ finds *a* cluster $\langle \pi_3, \mathcal{L}_3 \rangle = \mathcal{C}_{\mathcal{O}_i}(\pi_3)$ to which ℓ belongs (Sect. 4.2). Note that the choice of cluster is arbitrary. The rules are applied on $\langle \pi_3, \mathcal{L}_3 \rangle$ if the required pre-conditions are met (Line 49 and Line 53, respectively). When applicable, SUBSUME returns a new lemma that is added to the frames, while CONJECTURE returns a new POB that is added to the queue. Note that the latter is a *may* POB, in the sense that some of the states it represents *may not* lead to safety violation.

Ensuring progress. SPACER always makes progress: as its search continues, it establishes absence of counterexamples of deeper and deeper depths. However, GSPACER does not ensure progress. Specifically, unrestricted application of the `Concretize` and `Conjecture` rules can make GSPACER diverge even on executions of a fixed bound. In our implementation, we ensure progress by allotting a fixed amount of *gas* to each pattern, π, that forms a cluster. Each time `Concretize` or `Conjecture` is applied to a cluster with π as the pattern, π loses some gas. Whenever π runs out of gas, the rules are no longer applied to any cluster with π as the pattern. There are finitely many patterns (assuming LIA terms are normalized). Thus, in each bounded execution of GSPACER, the `Concretize` and `Conjecture` rules are applied only a finite number of times, thereby, ensuring progress. Since the `Subsume` rule does not hinder progress, it is applied without any restriction on gas.

4.7 Global guidance for linear rational aritmetic

So far, we have described our implementation of global guidance rules for LIA. The implementation for LRA is very similar. In fact, procedure for clustering lemmas, the pre-conditions for applying the three global guidance rules, and the procedures to conjecture and concretize POBS carry over without any modifications. The only difference is in the implementation of the `Subsume` rule (Algorithm 4). For LIA, the procedure involves three main steps: computing convex hull (Line 8), adding divisibility constraints (Line 16), and eliminating auxiliary variables (Line 12). Since we cannot have divisibility constraints for rational variables, we skip this step when implementing `Subsume` for LRA. The rest of the algorithm remains the same.

5 Evaluation

We have implemented[2] GSPACER (Algorithm 6) as an extension to SPACER. To reduce the dimension of a matrix (in SUBSUME, Sect. 4.3), we compute pairwise linear dependencies between all pairs of columns instead of computing the full kernel. This does not necessarily reduce the dimension of the matrix to its rank, but, is sufficient for our benchmarks. We have experimented with computing the full kernel using SageMath [15], but the overall performance did not improve. Clustering is implemented by anti-unification. LIA and LRA terms are normalized using default Z3 simplifications.

[2] https://github.com/hgvk94/z3/tree/ggbranch.

Table 1 Number of LIA benchmarks in each category after removing duplicates

Category	# Benchmarks
CHC-COMP18-LIA	438
CHC-COMP19-LIA-LIN	325
CHC-COMP19-LIA-nonLIN	283
CHC-COMP20-LIA-LIN	428
CHC-COMP20-LIA-nonLIN	416
CHC-COMP21-LIA-LIN	417
CHC-COMP21-LIA-nonLIN	432

Table 2 Number of LRA benchmarks in each category after removing duplicates

Category	# Benchmarks
CHC-COMP18	132
CHC-COMP19	244
CHC-COMP20	499

We used 3 sets of benchmarks for our evaluation: LIA and LRA instances from CHC-COMP 2018–21 [3] and benchmarks used in the evaluation of [6]. The CHC-COMP benchmarks have been curated from a variety of applications including Linux driver verification, synthesis and higher order program verification. The LIA benchmarks contain linear and non-linear CHCs whereas the LRA benchmarks only contain linear CHCs. We removed all duplicates from the benchmark suite. Since the LRA benchmarks in CHC-COMP 2021 are the same as those in CHC-COMP 2020, we omit them in our evaluation. The numbers of instances in each category, after removing duplicates, are shown in Tables 1 and 2.

To evaluate our implementation, we have conducted 3 sets of experiments. All experiments were run on Intel E5-2690 V2 CPU at 3GHz with 128GB memory with a timeout of 10 minutes. The first set of experiments (Sect. 5.1) evaluate the performance of local reasoning with global guidance against pure local reasoning. The second set of experiments (Sect. 5.2) compare the performance of local reasoning with global guidance to solely global reasoning. The third set of experiments (Sect. 5.3) compare the effectiveness of each of the global guidance rules.

We use Spacer as a *baseline* for pure local reasoning because it dominated CHC-COMP by solving 85% of the LIA benchmarks in CHC-COMP 2019 (20% more than the runner up) and 60% of the LIA benchmarks in CHC-COMP 2018 (10% more than runner up). In CHC-COMP 2020 and CHC-COMP 2021, GSpacer participated instead of Spacer and won all LIA tracks. Our evaluation shows that GSpacer outperforms Spacer both in terms of number of solved instances and, more importantly, in overall robustness.

5.1 Comparison with Spacer

Comparison on LIA *instances* Table 3 summarizes the comparison between Spacer and GSpacer on LIA instances from CHC-COMP. Since both tools can use a variety of interpolation strategies during lemma generalization (Line 45 in Algorithm 6), we compare three different configurations of each: *bw* and *fw* stand for two interpolation strategies, *backward* and *forward*, respectively, already implemented in Spacer, and *sc* stands for turning

Table 3 Comparison between SPACER and GSPACER on LIA benchmarks from CHC-COMP

Bench	GSPACER						SPACER						VBS	
	sc		bw		fw		sc		bw		fw			
	Safe	Unsafe	Safe	Unsafe	Safe	Unsafe	Safe	Unsafe	Safe	Unsafe	Safe	Unsafe	Safe	Unsafe
CHC18-Lia	208	55	**210**	55	205	52	115	61	149	**64**	142	62	223	73
CHC19-Lia-Lin	168	79	**169**	77	167	79	110	79	162	**81**	165	80	182	82
CHC19-Lia-NonLin	**156**	**116**	153	114	153	114	136	**116**	145	**116**	146	115	158	116
CHC20-Lia-Lin	223	133	**227**	132	224	134	190	135	209	**136**	209	135	230	136
CHC20-Lia-NonLin	203	200	**205**	**203**	204	201	196	**203**	202	**203**	201	202	207	203
CHC21-Lia-Lin	198	87	199	90	**200**	88	141	90	175	**92**	177	**92**	205	95
CHC21-Lia-NonLin	251	134	**254**	129	251	127	225	**138**	236	134	230	133	262	139
Total	1407	804	**1417**	800	1404	795	1113	822	1278	**826**	1270	819	1467	844

The bold numbers denote the best performing tool in each category

interpolation off and generalizing lemmas only by *subset clauses* computed by inductive generalization. VBS stands for the Virtual Best Solver.

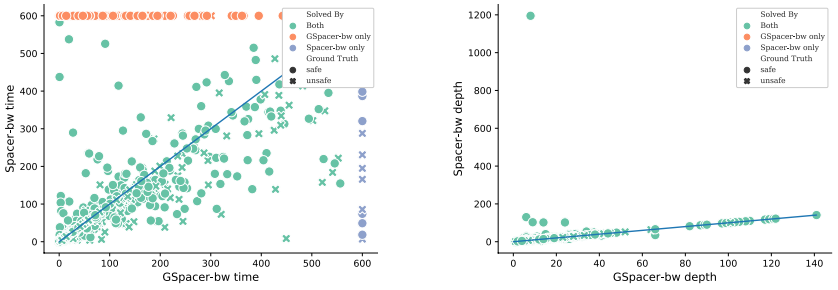
Any configuration of GSpacer solves significantly more instances than even the best configuration of Spacer. Figure 3 provides a more detailed comparison between the best configurations of both tools in terms of running time and depth of convergence. There is no clear trend in terms of running time on instances solved by both tools. This is not surprising—SMT-solving run time is highly non-deterministic and any change in strategy has a significant impact on performance of SMT queries involved. In terms of depth, it is clear that GSpacer converges at the same or lower depth. The depth is significantly lower for instances solved only by GSpacer.

Moreover, the performance of GSpacer is not significantly affected by the interpolation strategy used. In fact, the configuration *sc* in which interpolation is disabled performs the best in non-linear instances from CHC-COMP 2019, and only slightly worse in other categories. In comparison, disabling interpolation hurts Spacer significantly.

Comparison on LRA *benchmarks* To see how well the global guidance rules scale across theories, we compared GSpacer with Spacer on LRA benchmarks from CHC-COMP. The results are summarized in Table 4. The meanings of the columns are the same as in Table 3.

Most significantly, we find that global guidance is robust. While Spacer has a noticeable drop in performance when interpolation is turned off, GSpacer does not. Second, while the number of instances solved by GSpacer is similar to the number of instances solved by Spacer, the instances solved are different. In particular, the VBS significantly outperforms both solvers. Finally, the improvement in GSpacer on LRA benchmarks is not as significant as on LIA benchmarks. We believe this is a property of the benchmark set rather than the intrinsic difference between LIA and LRA. We conjecture that additional global guidance rules, specifically ones taking advantage of symmetry in the lemmas, will boost the performance of GSpacer.

Comparison on unsafe instances Spacer consistently solves more unsafe instances than GSpacer. This is an expected side effect of the global guidance rules. GSpacer explores different ways to find an inductive invariant and, on unsafe instances, this exploration is completely wasteful. This exploration comes at a cost. Each pob that global guidance generates is checked for reachability and, if blocked, generalized using inductive generalization. Both these are one or more queries to an SMT solver. The cost of exploration is checked using



(a) running time (b) depth explored

Fig. 3 Best configurations on LIA benchmarks: GSpacer versus Spacer

Table 4 Comparing GSpacer with Spacer on LRA benchmarks from CHC-COMP

Bench	GSpacer						Spacer						VBS	
	sc		bw		fw		sc		bw		fw			
	Safe	Unsafe	Safe	Unsafe	Safe	Unsafe	Safe	Unsafe	Safe	Unsafe	Safe	Unsafe	Safe	Unsafe
CHC18	55	**8**	51	7	**57**	7	56	**8**	54	**8**	55	**8**	62	8
CHC19	110	**47**	112	45	**113**	45	78	**47**	101	46	101	46	123	47
CHC20	**215**	73	214	74	210	74	186	**75**	199	74	207	74	239	75
Total	**380**	128	377	126	**380**	126	320	**130**	354	128	363	128	424	130

The bold numbers denote the best performing tool in each category

gas (Sect. 4.6). For each cluster, gas is initially set to a constant value and decreased each time a global guidance rule is applied. Once a cluster runs out of gas, global guidance rules are not applied to the cluster. While this heuristic reduces exploration, it does not reduce it to zero, hence the deterioration in performance.Figure 4 provides a detailed comparison of GSpacer with and without interpolation. Interpolation makes no difference to the depth of convergence. This implies that lemmas that are discovered by interpolation are discovered as efficiently by the global rules of GSpacer. On the other hand, interpolation significantly increases the running time. Interestingly, the time spent in interpolation itself is insignificant. However, the lemmas produced by interpolation tend to slow down other aspects of the algorithm. Most of the slow down is in increased time for inductive generalization and in computation of predecessors. The comparison between the other interpolation-enabled strategy and GSpacer-*sc* shows a similar trend.We used the ML-based data-driven invariant inference tool LinearArbitrary [6] as a baseline for purely global reasoning. Compared to other similar approaches, LinearArbitrary stands out by supporting invariants with arbitrary Boolean structure over arbitrary linear predicates. It is completely automated and does not require user-provided predicates, grammars, or any other guidance. For the comparison with LinearArbitrary, we have used both the CHC-COMP benchmarks, as well as the benchmarks from the artifact evaluation of [6]. The machine and timeout remain the same. Our evaluation shows that GSpacer is superior in this case as well.

5.2 Comparison with LinearArbitrary

In [6], the authors show that LinearArbitrary, to which we refer as LArb for short, significantly outperforms Spacer on a curated subset of benchmarks from SV-COMP [16] competition.

We first compare LArb against GSpacer on CHC-COMP instances. Table 5 shows the number of instances solved by both solvers. We see that LArb solves way fewer instances than even baseline Spacer.

For a more meaningful comparison, we compared Spacer, LArb, and GSpacer on the benchmarks from the artifact evaluation of [6]. The results are summarized in Table 6. As expected, LArb outperforms the baseline Spacer on the safe benchmarks. On unsafe benchmarks, Spacer is significantly better than LArb. In both categories, GSpacer dominates solving more safe benchmarks than either Spacer or LArb, while matching performance of Spacer on unsafe instances. Furthermore, GSpacer remains orders of magnitude faster than LArb on benchmarks that are solved by both. This comparison shows that incorporating local reasoning with global guidance not only mitigates its shortcomings but also surpasses global data-driven reasoning.

5.3 Effectiveness of individual global guidance rules

Table 7 summarizes the effectiveness of individual global guidance rules on CHC-COMP LIA benchmarks. Each column in Table 7 corresponds to running GSpacer with just one of the three global guidance rules enabled. We see that the Subsume configuration (that is, running GSpacer with just the `Subsume` rule) solves many more instances than either of the other configurations. However, even the `Subsume` rule individually does not solve merely as many instances as GSpacer with all three rules combined. Figure 5 compares the

Table 5 Comparison with LAR using CHC-COMP benchmarks

Bench	SPACER		LARB		GSPACER	
	Safe	Unsafe	Safe	Unsafe	Safe	Unsafe
CHC-18-LIA	149	**64**	64	8	**210**	55
CHC-19-LIA-LIN	162	**81**	131	18	**169**	77

The bold numbers denote the best performing tool in each category

Table 6 Comparison with LAR

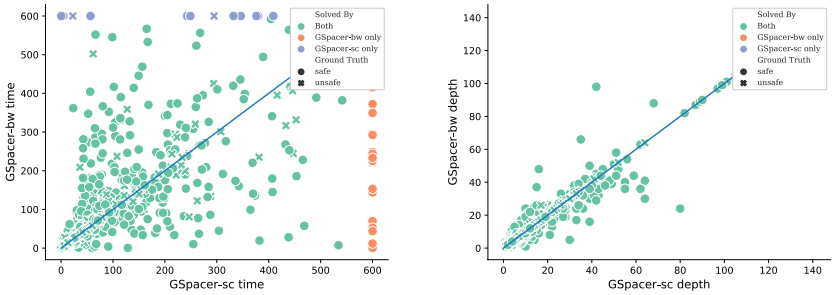
Bench	SPACER		LARB		GSPACER		VBS	
	Safe	Unsafe	Safe	Unsafe	Safe	Unsafe	Safe	Unsafe
PLDI18	215	**68**	270	65	**280**	67	286	68

The bold numbers denote the best performing tool in each category

Table 7 Effectiveness of individual rules of GSPACER on CHC-COMP LIA instances

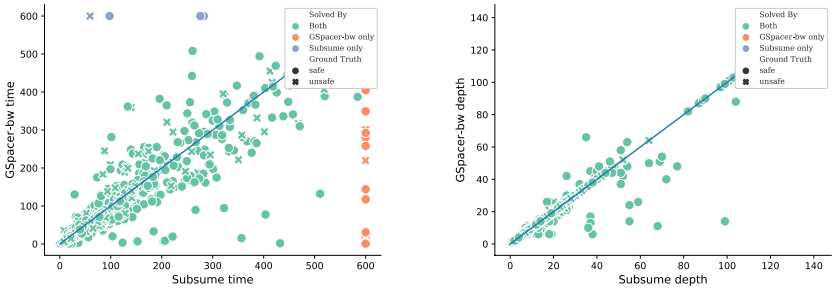
Bench	Conjecture		Concretize		Subsume		VBS	
	Safe	Unsafe	Safe	Unsafe	Safe	Unsafe	Safe	Unsafe
CHC18-Lia	146	52	149	**54**	**191**	51	208	56
CHC19-Lia-Lin	160	77	161	76	**165**	**79**	168	79
CHC19-Lia-NonLin	144	114	144	**115**	**152**	114	154	115
CHC20-Lia-Lin	209	132	212	133	**222**	**134**	226	134
CHC20-Lia-NonLin	201	201	201	200	**204**	**202**	205	202
CHC21-Lia-Lin	172	**88**	174	87	**193**	**88**	200	89
CHC21-Lia-NonLin	234	**132**	233	**132**	**250**	130	254	132
Total	1266	796	1274	797	**1377**	**798**	1415	807

The bold numbers denote the best performing tool in each category



(a) running time (b) depth explored

Fig. 4 Comparing GSPACER with different interpolation tactics on LIA benchmarks



(a) running time (b) depth explored

Fig. 5 Comparing GSpacer with just the `Subsume` rule (x-axis) against GSpacer with all three rules together (y-axis)

running time and depth of convergence of Subsume configuration (x-axis) against those of the full GSpacer (y-axis). Clearly, there are many instances in which GSpacer with all three rules converges at a significantly lower depth than the GSpacer with just the `Subsume` rule.

6 Related work

The limitations of local reasoning in SMT-based infinite state model checking are well known. Most commonly, they are addressed with either (a) different strategies for local generalization in interpolation (e.g., [17–20]), or (b) shifting the focus to *global* invariant inference by learning an invariant of a restricted shape (e.g., [4–7, 21]).

Interpolation strategies. Albarghouthi and McMillan [18] suggest to minimize the number of literals in an interpolant, arguing that simpler (i.e., fewer half-spaces) interpolants are more likely to generalize. This helps with myopic generalizations (Fig. 1a), but not with excessive generalizations (Fig. 1b). On the contrary, Blicha et al. [19] decompose interpolants to be numerically simpler (but with more literals), which helps with excessive, but not with myopic, generalizations. Deciding *locally* between these two techniques or on their combination (i.e., some parts of an interpolant might need to be split while others combined) seems impossible. Schindler and Jovanovic [20] propose local interpolation that bounds the number of lemmas generated from a single POB (which helps with Fig. 1c), but only if inductive generalization is disabled. Finally, [17] suggests using external guidance, in a form of predicates or terms, to guide interpolation. In contrast, GSpacer uses global guidance, based on the current proof, to direct different local generalization strategies. Thus, the guidance is automatically tuned to the specific instance at hand rather than to a domain of problems.

Global invariant inference. An alternative to inferring lemmas for the inductive invariant by blocking counterexamples is to enumerate the space of potential candidate invariants [4–7, 21]. This does not suffer from the pitfall of local reasoning. However, it is only effective when the search space is constrained. While these approaches perform well on their

target domain, they do not generalize well to a diverse set of benchmarks, as illustrated by results of CHC-COMP and our empirical evaluation in Sect. 5.

Locality in SMT and IMC . Local reasoning is also a known issue in SMT, and, in particular, in DPLL(T) (e.g., [22]). However, we are not aware of global guidance techniques for SMT solvers. Interpolation-based Model Checking (IMC) [23, 24] that uses interpolants from proofs, inherits the problem. Compared to IMC, the propagation phase and inductive generalization of IC3 [1], can be seen as providing global guidance using lemmas found in other parts of the search-space. In contrast, GSpacer magnifies such global guidance by exploiting patterns within the lemmas themselves.

IC3-SMT-based Model Checkers. There are a number of IC3-style SMT-based infinite state model checkers, including [2, 25, 26]. To our knowledge, none extend the IC3-SMT framework with a global guidance. A rule similar to `Subsume` is suggested in [27] for the theory of bit-vectors and in [28] for LRA, but in both cases without global guidance. In [28], it is implemented via a combination of syntactic closure with interpolation, whereas we use MBP instead of interpolation. Refinement State Mining in [29] uses similar insights to our `Subsume` rule to refine predicate abstraction.

7 Conclusion and future work

This paper introduces *global guidance* to mitigate the limitations of the local reasoning performed by SMT-based IC3-style model checking algorithms. Global guidance is necessary to redirect such algorithms from divergence due to persistent local reasoning. To this end, we present three general rules that introduce new lemmas and pobs by taking a global view of the lemmas learned so far. The new rules are not theory-specific, and, as demonstrated by Algorithm 6, can be incorporated to IC3-style solvers without modifying existing architecture. We instantiate, and implement, the rules for LIA and LRA in GSpacer, which extends Spacer.

Our evaluation shows that global guidance brings significant improvements to local reasoning, and surpasses invariant inference based solely on global reasoning. More importantly, global guidance decouples Spacer 's dependency on interpolation strategy and performs almost equally well under all three interpolation schemes we consider. As such, using global guidance in the context of theories for which no good interpolation procedure exists arises as a promising direction for future research. For example, [30] proposes global guidance for bit-vectors. Other theories that could benefit from global guidance include algebraic datatypes and arrays.

Acknowledgements GSpacer is openly available at https://github.com/hgvk94/z3/tree/ggbranch. The benchmarks used for our evaluation are openly available at https://github.com/chc-comp. The raw evaluation results are available at https://hgvk94.github.io/gspacer. We thank Xujie Si for running the LArb experiments and collecting results. We thank the ERC starting Grant SYMCAR 639270 and the Wallenberg Academy Fellowship TheProSE for supporting the research visit. The research leading to these results has received funding from the European Research Council under the European Union's Horizon 2020 research and innovation programme (grant agreement No [759102-SVIS]). This research was partially supported by the United States-Israel Binational Science Foundation (BSF) grant No. 2016260, and the Israeli Science Foundation (ISF) grant No. 1810/18. This research was partially supported by grants from Natural Sciences and Engineering Research Council Canada.

References

1. Bradley AR (2011) SAT-based model checking without unrolling. In: Verification, model checking, and abstract interpretation - 12th international conference, VMCAI 2011, Austin, TX, USA, January 23–25, 2011. Proceedings. pp 70–87 (2011). https://doi.org/10.1007/978-3-642-18275-4_7
2. Komuravelli A, Gurfinkel A, Chaki S (2014) SMT-based model checking for recursive programs. In: Computer aided verification - 26th international conference, CAV 2014, held as part of the vienna summer of logic, VSL 2014, Vienna, Austria, July 18–22, 2014. Proceedings. pp 17–34. https://doi.org/10.1007/978-3-319-08867-9_2
3. CHC-COMP, CHC-COMP. https://chc-comp.github.io
4. Flanagan C, Leino KRM (2001) Houdini, an annotation assistant for ESC/Java. In: FME 2001: formal methods for increasing software productivity, international symposium of formal methods Europe, Berlin, Germany, March 12–16, 2001, proceedings. pp 500–517. https://doi.org/10.1007/3-540-45251-6_29
5. Fedyukovich G, Kaufman SJ, Bodík R (2017) Sampling invariants from frequency distributions. In: 2017 formal methods in computer aided design, FMCAD 2017, Vienna, Austria, October 2–6, 2017. pp 100–107. https://doi.org/10.23919/FMCAD.2017.8102247
6. Zhu H, Magill S, Jagannathan S (2018) A data-driven CHC solver. In: Proceedings of the 39th ACM SIGPLAN conference on programming language design and implementation, PLDI 2018, Philadelphia, PA, USA, June 18–22, 2018. pp 707–721. https://doi.org/10.1145/3192366.3192416
7. Garg P, Neider D, Madhusudan P, Roth D (2016) Learning invariants using decision trees and implication counterexamples. In: Proceedings of the 43rd annual ACM SIGPLAN-SIGACT symposium on principles of programming languages, POPL 2016, St. Petersburg, FL, USA, January 20–22, 2016. pp 499–512. https://doi.org/10.1145/2837614.2837664
8. de Moura LM, Bjørner N (2008) Z3: An efficient SMT solver. In: Tools and algorithms for the construction and analysis of systems, 14th international conference, TACAS 2008, held as part of the joint european conferences on theory and practice of software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings. pp 337–340. https://doi.org/10.1007/978-3-540-78800-3_24
9. Krishnan HGV, Chen Y, Shoham S, Gurfinkel A (2020) Global guidance for local generalization in model checking. In: Lahiri SK, Wang C (eds) Computer aided verification - 32nd international conference, CAV 2020, Los Angeles, CA, USA, July 21–24, 2020, proceedings, part II. Lecture notes in computer science, vol 12225. Springer, pp 101–125. https://doi.org/10.1007/978-3-030-53291-8_7
10. Bjørner N, Janota M (2015) Playing with quantified satisfaction. In: 20th international conferences on logic for programming, artificial intelligence and reasoning - short presentations, LPAR 2015, Suva, Fiji, November 24–28, 2015. pp 15–27. http://www.easychair.org/publications/paper/252316
11. Cousot P, Cousot R (1977) Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Conference record of the fourth ACM symposium on principles of programming languages, Los Angeles, California, USA, January 1977. pp 238–252. https://doi.org/10.1145/512950.512973
12. Bulychev PE, Kostylev EV, Zakharov VA (2009) Anti-unification algorithms and their applications in program analysis. In: Perspectives of systems informatics, 7th international andrei ershov memorial conference, PSI 2009, Novosibirsk, Russia, June 15–19, 2009. Revised papers. pp 413–423. https://doi.org/10.1007/978-3-642-11486-1_35
13. Yernaux G, Vanhoof W (2019) Anti-unification in constraint logic programming. TPLP 19(5–6):773–789. https://doi.org/10.1017/S1471068419000188
14. Benoy F, King A, Mesnard F (2005) Computing convex hulls with a linear solver. TPLP 5(1–2):259–271. https://doi.org/10.1017/S1471068404002261
15. The Sage Developers (2017) SageMath, the Sage mathematics software system (Version 8.1.0). https://www.sagemath.org
16. SV-COMP: SV-COMP. https://sv-comp.sosy-lab.org/
17. Leroux J, Rümmer P, Subotic P (2016) Guiding Craig interpolation with domain-specific abstractions. Acta Inf 53(4):387–424. https://doi.org/10.1007/s00236-015-0236-z

18. Albarghouthi A, McMillan KL (2013) Beautiful interpolants. In: Computer aided verification - 25th international conference, CAV 2013, Saint Petersburg, Russia, July 13–19, 2013. Proceedings. pp 313–329. https://doi.org/10.1007/978-3-642-39799-8_22
19. Blicha M, Hyvärinen AEJ, Kofron J, Sharygina N (2019) Decomposing farkas interpolants. In: Tools and algorithms for the construction and analysis of systems - 25th international conference, TACAS 2019, held as part of the European joint conferences on theory and practice of software, ETAPS 2019, Prague, Czech Republic, April 6–11, 2019, proceedings, part I. pp 3–20. https://doi.org/10.1007/978-3-030-17462-0_1
20. Schindler T, Jovanovic D (2018) Selfless interpolation for infinite-state model checking. In: Verification, model checking, and abstract interpretation - 19th international conference, VMCAI 2018, Los Angeles, CA, USA, January 7–9, 2018, proceedings. pp 495–515. https://doi.org/10.1007/978-3-319-73721-8_23
21. Champion A, Chiba T, Kobayashi N, Sato R (2018) ICE-based refinement type discovery for higher-order functional programs. In: Tools and algorithms for the construction and analysis of systems - 24th international conference, TACAS 2018, held as part of the European joint conferences on theory and practice of software, ETAPS 2018, Thessaloniki, Greece, April 14–20, 2018, proceedings, part I. pp 365–384. https://doi.org/10.1007/978-3-319-89960-2_20
22. McMillan KL, Kuehlmann A, Sagiv M (2009) Generalizing DPLL to richer logics. In: Computer aided verification, 21st international conference, CAV 2009, Grenoble, France, June 26–July 2, 2009. Proceedings. pp 462–476. https://doi.org/10.1007/978-3-642-02658-4_35
23. McMillan KL (2003) Interpolation and SAT-based model checking. In: Computer aided verification, 15th international conference, CAV 2003, Boulder, CO, USA, July 8–12, 2003, proceedings. pp 1–13. https://doi.org/10.1007/978-3-540-45069-6_1
24. McMillan KL (2006) Lazy abstraction with interpolants. In: computer aided verification, 18th international conference, CAV 2006, Seattle, WA, USA, August 17–20, 2006, proceedings. pp 123–136. https://doi.org/10.1007/11817963_14
25. Jovanovic D, Dutertre B (2016) Property-directed k-induction. In: 2016 formal methods in computer-aided design, FMCAD 2016, Mountain View, CA, USA, October 3–6, 2016. pp 85–92. https://doi.org/10.1109/FMCAD.2016.7886665
26. Cimatti A, Griggio A, Mover S, Tonetta S (2016) Infinite-state invariant checking with IC3 and predicate abstraction. Formal Methods Syst Des 49(3):190–218. https://doi.org/10.1007/s10703-016-0257-4
27. Welp T, Kuehlmann A (2013) QF_BV model checking with property directed reachability. In: Design, automation and test in Europe, DATE 13, Grenoble, France, March 18–22, 2013. pp 791–796. https://doi.org/10.7873/DATE.2013.168
28. Bjørner N, Gurfinkel A (2015) Property directed polyhedral abstraction. In: Verification, model checking, and abstract interpretation - 16th international conference, VMCAI 2015, Mumbai, India, January 12–14, 2015. Proceedings. pp 263–281. https://doi.org/10.1007/978-3-662-46081-8_15
29. Birgmeier J, Bradley AR, Weissenbacher G (2014) Counterexample to induction-guided abstraction-refinement (CTIGAR). In: Computer aided verification - 26th international conference, CAV 2014, held as part of the Vienna summer of logic, VSL 2014, Vienna, Austria, July 18–22, 2014. Proceedings. pp 831–848. https://doi.org/10.1007/978-3-319-08867-9_55
30. K HGV, Fedyukovich G, Gurfinkel A (2020) Word level property directed reachability. In: Proceedings of the 39th international conference on computer-aided design. ICCAD '20. Association for computing machinery, New York, NY, USA. https://doi.org/10.1145/3400302.3415708